Michael Morrison

Learn

the skills and concepts to create and deploy J2ME applications in just 21 days

Apply

your knowledge in the real world

SAMS Teach Yourself Wireless Java™ with J2ME in 21 Days

SAMS

Michael Morrison

SAMS Teach Yourself

Wireless Java with J2ME in 21 Days

SAMS
201 West 103rd St., Indianapolis, Indiana, 46290 USA

Sams Teach Yourself Wireless Java with J2ME in 21 Days

International Standard Book Number: 0-672-32142-4

Library of Congress Catalog Card Number: 2001089623

Printed in the United States of America

First Printing: June 2001

04 03 02 01 4 3 2 1

EXECUTIVE EDITOR
Michael Stephens

ACQUISITIONS EDITOR
Carol Ackerman

DEVELOPMENT EDITOR
Bryan Morgan

MANAGING EDITOR
Matt Purcell

PROJECT EDITOR
Andy Beaster

COPY EDITORS
Cindy Fields
Gene Redding

INDEXER
Johnna Dinse

PROOFREADER
Candice Hightower

TECHNICAL EDITOR
Bryan Morgan

TEAM COORDINATOR
Lynne Williams

MEDIA DEVELOPER
Dan Scherf

INTERIOR DESIGNER
Gary Adair

COVER DESIGNER
Aren Howell

PAGE LAYOUT
Susan Geiselman

Contents at a Glance

Introduction 1

Part I Getting Started with J2ME and MIDP 7

Day 1 Java 2 Micro Edition: The Big Picture 9

Day 2 Assembling a J2ME Development Kit 25

Day 3 Getting to Know the J2ME Emulator 41

Day 4 Building Your First MIDlet 63

Day 5 Working within the CLDC and MIDP APIs 85

Day 6 Creating Custom Device Profiles 105

Part II Digging Deeper into MIDlet Development 125

Day 7 Building Graphical MIDlets 127

Day 8 Making the Most of MIDlet GUIs 151

Day 9 Exploring MIDlet I/O and Networking 189

Day 10 Being Your Own Meteorologist 213

Day 11 Wireless Navigation 237

Day 12 Optimizing MIDlet Code 265

Part III Wireless Information Management with MIDlets 279

Day 13 Using the MIDP Record Management System (RMS) 281

Day 14 Staying in Touch with Your Contacts 311

Day 15 Managing Your Finances 339

Day 16 Bidding on the Go 373

Part IV Entertaining Without the Wires 401

Day 17 Creating Animated MIDlets 403

Day 18 An Ode to Pong: Creating MIDlet Gaming 437

Day 19 The Next Level of MIDlet Gaming 461

Part V Exploring J2ME Wireless Technologies 491

Day 20 A J2ME Wireless Device Tour 493

Day 21 The Future of J2ME 509

Part VI Appednixes **521**

A Quiz Answers 523

B What's on the CD-ROM 531

Index 535

Contents

Introduction **1**

Part I Getting Started with J2ME and MIDP **7**

Day 1 Java 2 Micro Edition: The Big Picture **9**

Java Without Wires 10
What is J2ME? 13
J2ME and Other Wireless Technologies 14
  J2ME and WAP 15
  J2ME and SMS 15
  J2ME and Bluetooth 16
Configurations and the CLDC 16
  The CLDC and the Java Programming Language 18
  The CLDC and Security 19
The MID Profile (MIDP) 19
  Hardware 20
  Software 21
Summary 22
Q&A 23
Workshop 23
  Quiz 23
  Exercises 23

Day 2 Assembling a J2ME Development Kit **25**

A J2ME Development Primer 26
Inside the J2ME Wireless Toolkit 29
  The Bytecode Verifier 30
  The J2ME Emulator 31
  The Configuration Editor 32
Visual J2ME Development Environments 33
  KToolBar 33
  Forte for Java 34
  CodeWarrior for Java 36
  JBuilder Handheld Express 36
J2ME Web Servers 36
Summary 37
Q&A 37
Workshop 38
  Quiz 38
  Exercises 38

Day 3 Getting to Know the J2ME Emulator 41

J2ME Emulator Essentials 42
Emulating Physical Devices 44
Running the Emulator from the Command Line 45
  The J2ME Wireless Toolkit 46
  The Motorola SDK for J2ME 51
Running the Emulator Within a Visual Environment 56
  KToolBar 56
  Forte for Java 58
Summary 60
Q&A 61
Workshop 61
  Quiz 62
  Exercises 62

Day 4 Building Your First MIDlet 63

What Is a MIDlet? 64
Inside a MIDlet 65
  MIDlet Life Cycle 66
  MIDlet Commands 66
  The Display, Screens, and Canvases 66
MIDlet Development Revisited 67
A Painfully Simple MIDlet 68
  Coding the MIDlet 69
  Compiling the MIDlet 74
  Pre-Verifying the MIDlet 75
  Packaging the MIDlet 75
  Testing the MIDlet 78
Building and Testing the Howdy MIDlet with KToolBar 79
Summary 81
Q&A 82
Workshop 82
  Quiz 82
  Exercises 83

Day 5 Working within the CLDC and MIDP APIs 85

MIDlet Programming: A Tale of Two APIs 86
The CLDC API 87
  CLDC and the Standard J2SE API 88
  The CLDC Generic Connection Framework 91
The MIDP API 92
  MIDP and the Standard J2SE API 92
  MIDP-Specific Classes and Interfaces 93

Putting the APIs to Use 97
  Data Type Wrappers 97
  The Runtime Environment 98
  Date and Time 98
The SysInfo MIDlet 99
Summary 103
Q&A 103
Workshop 104
  Quiz 104
  Exercises 104

Day 6 Creating Custom Device Profiles 105

Understanding Device Profiles 106
Using the Configuration Editor 107
  Setting the Display Properties and Key Mappings 108
  Establishing the Available Memory and Execution Speed 115
  Determining the Networking Capabilities 116
  Supporting Events 117
  Finishing Up the Device Profile 117
Tweaking a Device Profile 118
Testing a Custom Device Profile 120
Summary 122
Q&A 122
Workshop 123
  Quiz 123
  Exercises 123

Part II Digging Deeper into MIDlet Development 125

Day 7 Building Graphical MIDlet 127

The MIDP Graphics Coordinate System 128
A Quick Color Primer 129
The Graphics Class 131
  Drawing Graphics Primitives 132
  Drawing Text 135
  Drawing Images 138
Building Your First Graphical MIDlet 139
Building a Slide Show MIDlet 143
Summary 148
Q&A 149
Workshop 149
  Quiz 149
  Exercises 150

Day 8 Making the Most of MIDlet GUIs 151

Getting to Know the Display 152
Working with Screens and Forms 154
Assessing the MIDP GUI Components 158
 The `Alert` Class 159
 The `List` Class 160
 The `TextBox` Class 163
 The `Ticker` Class 164
 The `StringItem` Class 165
 The `ImageItem` Class 167
 The `TextField` Class 169
 The `DateField` Class 170
 The `ChoiceGroup` Class 172
 The `Gauge` Class 173
Using Commands 175
The Mortgage Sample MIDlet 178
 The User Interface 178
 Handling Commands 180
 Crunching the Numbers 182
 Putting It All Together 182
Summary 186
Q&A 187
Workshop 187
 Quiz 187
 Exercises 188

Day 9 Exploring MIDlet I/O and Networking 189

Inside MIDP Networking 190
Understanding MIDlet I/O 193
 The `InputStream` Class 194
 The `OutputStream` Class 195
Communicating Across Network Connections 196
 Making the Connection 197
 Obtaining an Input Stream 197
 Reading from the Stream 197
The Fortune Example MIDlet 198
 The User Interface 200
 Handling Commands 202
 Retrieving and Showing the Fortunes 202
 Putting It All Together 205
Testing the Fortune MIDlet 208
Summary 209

Q&A 210
Workshop 210
Quiz 210
Exercises 211

Day 10 Being Your Own Wireless Meteorologist 213

Weather and the Web 214
Peeking Inside a Weather Web Page 217
Constructing the Weather MIDlet 221
The User Interface 222
Handling Commands 224
Retrieving Weather Data 226
Putting It All Together 228
Testing the Weather MIDlet 232
Summary 233
Q&A 233
Workshop 234
Quiz 234
Exercises 235

Day 11 Wireless Navigation 237

Using the Web to Get From A to B 238
Ripping Apart a Driving Directions Web Page 243
Constructing the Directions MIDlet 248
The User Interface 248
Handling Commands 251
Prepping the Location Information 252
Retrieving Directions Data 253
Putting It All Together 256
Testing the Directions MIDlet 259
Summary 261
Q&A 262
Workshop 262
Quiz 262
Exercises 263

Day 12 Optimizing MIDlet Code 265

MIDlet Optimization Basics 266
Optimizing for Maintainability 267
Optimizing for Size 267
Optimizing for Speed 268

MIDlet Optimization Tips 268
Simplify Your MIDlets 269
Minimize MIDlet Data 269
Reduce MIDlet Memory Usage 269
Java Optimization Techniques 271
Eliminate Unnecessary Evaluations 272
Eliminate Common Subexpressions 273
Take Advantage of Local Variables 273
Expand Loops 273
Putting MIDlet Optimization into Perspective 274
Summary 275
Q&A 276
Workshop 276
Quiz 276
Exercises 276

Part III Wireless Information Management with MIDlets 279

Day 13 Using the MIDP Record Management System (RMS) 281

What Is the MIDP Record Management System (RMS)? 282
Inside the RMS Package 283
The `RecordStore` Class 284
The RMS Interfaces 284
Working with Record Stores 285
Opening a Record Store 285
Adding New Records 286
Getting Records 287
Deleting Records 287
Enumerating Records 288
Closing the Store 289
The ToDoList Sample MIDlet 289
Managing the Task Records 290
The User Interface 295
Handling Commands 298
Cleaning Up 300
Putting It All Together 300
Summary 307
Q&A 308
Workshop 308
Quiz 308
Exercises 309

Day 14 Staying in Touch with Your Contacts 311

Managing Contacts on the Go 312
Constructing the Contacts MIDlet 313
Managing the Contact Records 314
The User Interface 320
Handling Commands 323
Putting It All Together 326
Testing the Contacts MIDlet 331
Summary 335
Q&A 336
Workshop 337
Quiz 337
Exercises 337

Day 15 Managing Your Finances 339

Tracking Your Mobile Spending 340
Building the CheckRegister MIDlet 341
Managing the Transaction Records 341
Assessing the Member Variables 349
Inside the Constructor 351
Handling Commands 353
Putting It All Together 359
Testing the CheckRegister MIDlet 365
Summary 370
Q&A 370
Workshop 371
Quiz 371
Exercises 371

Day 16 Bidding on the Go 373

Keeping Up with Online Auctions from Afar 374
Constructing the AuctionWatch MIDlet 377
Managing the Item Records 377
The User Interface 381
Handling Commands 384
Retrieving the Live Auction Data 386
Putting It All Together 391
Testing the AuctionWatch MIDlet 396
Summary 399
Q&A 399
Workshop 399
Quiz 399
Exercises 400

Part IV Entertainment without the Wires 401

Day 17 Creating Animated MIDlets 403

What Is Animation? 404
Types of Animation 405
  Frame-based Animation 405
  Cast-based Animation 405
Implementing Sprite Animation 409
  The `Sprite` Class 410
  The `SpriteVector` Class 418
  The Background Classes 423
The Atoms Sample MIDlet 427
  The Animation Canvas 427
  Putting It All Together 432
Summary 433
Q&A 434
Workshop 434
  Quiz 434
  Exercises 435

Day 18 An Ode to Pong: Creating MIDlet Games 437

Designing Traveling Gecko 438
  Sprites 438
  Game Play 440
Constructing Traveling Gecko 440
  The Sprite Classes 442
  Inside the Traveling Gecko MIDlet 450
  The MIDlet Class 455
Testing Traveling Gecko 456
Summary 457
Q&A 458
Workshop 458
  Quiz 458
  Exercises 459

Day 19 The Next Level of MIDlet Gaming 461

A Crash Course in Game AI 462
Designing Connect4 465
  Game Play 466
  The User Interface 466
  AI Strategy 467
Constructing Connect4 469
  The Game Engine Classes 470
  Inside the Connect4 MIDlet 480

Testing Connect4 488
Summary 488
Q&A 489
Workshop 489
Quiz 489
Exercises 490

Part V Exploring J2ME Wireless Technologies 491

Day 20 A J2ME Wireless Tour 493

Exploring J2ME Devices 494
Java-Powered Mobile Phones 494
Pagers That Speak Java 497
Handheld Java Computers 499
Assessing the J2ME Application Landscape 502
Server Applications 502
Application Portals 503
End-User Applications 504
Summary 506
Q&A 506
Workshop 507
Quiz 507
Exercises 507

Day 21 The Future of J2ME 509

The CLDC Is Not Alone 510
The Convergence of J2ME and Other Java Technologies 511
The Demise of PersonalJava 512
Networking with Jini 512
Java Card and J2ME Smart Cards 513
J2ME: A Gadget Lover's Dream 514
Java In the Smart Home 514
Pumping Gas with J2ME 514
Java on Mars? 515
Gaming with J2ME 516
J2ME and Multimedia 517
Wireless Video 517
Musical Phones 517
Summary 518
Q&A 518
Workshop 519
Quiz 519
Exercises 519

Part VI Appendixes 521

A Quiz Answers 523

B What's on the CD-ROM 531

Sample Source Code 531
J2ME Tools 532

Index 535

About the Author

MICHAEL MORRISON is a writer, developer, toy inventor, and author of a variety of books including *The Unauthorized Guide to Pocket PCs* (Que Publishing, 2000), *Java In Plain English 3rd Edition* (IDG Books, 2000), *XML Unleashed* (Sams Publishing, 1999), and *Complete Idiot's Guide to Java 2* (Que Publishing, 1999). Michael is the instructor of several Web-based courses including DigitalThink's Introduction to Java 2 series, JavaBeans for Programmers series, and Win32 Fundamentals series. Michael also serves as a technical director for ReviewNet, a company that provides Web-based staffing tools for information technology personnel. Finally, Michael is the creative lead at Gas Hound Games, a toy company he co-founded that is located on the Web at `http://www.gashound.com/`.

When not risking life and limb on his skateboard or mountain bike, trying to avoid the penalty box in hockey, or watching movies with his wife, Masheed, Michael enjoys daydreaming next to his koi pond. You can visit Michael on the Web at `http://www.michaelmorrison.com/`. He also encourages you to check out his board game, Inc. The Game of Business, at `http://www.incthegame.com/`.

Dedication

For my friend Randy Weems, who taught me much of what I know about computers and technology. The techie world has lost one of its very few legitimate geniuses, but those of us who knew you well know that we've lost much more. We all miss you dearly.

Acknowledgments

Thanks to Mike Stephens for keeping my wife company at the Waterside Conference a while back, and for making this book a reality. A big thanks also goes to Carol Ackerman for one-upping all of the previously insane deadlines I've ever worked under and convincing me that I could work miracles, even though we all know better! Also, thanks to Bryan Morgan for keeping me on track as I digressed my way through the writing of this book.

An enormous thanks goes to my literary agent, Chris Van Buren, who somehow manages to keep me busy while hiding out in South America.

I am grateful to Jose Lacal and the rest of the folks at Motorola's iDEN Group for being such a powerful technical resource throughout the development of this book.

I am blessed to be surrounded by wonderful friends and family. Thanks to Travis and Sarah, Keith, Josh, Chuck and Coco, Cathy, Randy and Melody, and last but obviously not least, my incredible parents. Thank you all for being such positive influences in my life.

Of course, the biggest thanks of all goes to my wife, Masheed, who makes my life a pleasure to live.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Associate Publisher for Sams, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax:	317-581-4770
E-mail:	`feedback@samspublishing.com`
Mail:	Michael Stephens, Associate Publisher Sams 201 West 103rd Street Indianapolis, IN 46290 USA

Introduction

Lately there has been much talk about how the world is moving toward a future without wires. At last check, wires still deliver power to my house, and entirely too many wires form a tangled mess behind my computer. We are destined to live with at least some wires for quite some time. However, the wires that are likely to disappear from our lives are the ones that enable us to communicate with each other. The rapid adoption of mobile phones is the most obvious sign that people prefer the freedom of carrying around wireless devices to communicate more freely. Mobile phones are now such a part of everyday life that many states have passed legislation to limit or ban their use when driving. When politicians are aware of a new technology, it isn't very new any more!

The push from a wired world to a wireless one is steadily taking place, and will have a dramatic impact on how we access information and communicate with each other. Not surprisingly, wireless access to the Internet is getting the most attention when it comes to the benefits of going wireless. Wireless Internet access opens all kinds of opportunities for doing interesting things on mobile devices such as phones and pagers. Not only will you be able to browse specially formatted Web content, you will also be able to schedule flights, monitor auctions, and get stock quotes—tasks that are regularly performed on a wired desktop computer. A few mobile phones and pagers that enable you to do some of these things have been around for a little while, but they are somewhat limited in that there is no way to build entirely custom applications. The one thing missing from the wireless equation, until now, is a solid software platform on which to build compelling wireless applications.

Java 2 Micro Edition, or J2ME, dramatically changes the face of wireless mobile computing by making it possible to develop special Java programs (MIDlets) that are expressly designed to meet the needs of wireless mobile devices. Along with J2SE (Java 2 Standard Edition) and J2EE (Java 2 Enterprise Edition), J2ME completes the Java landscape with a platform for mobile computing. Perhaps the most exciting thing about J2ME is that it isn't simply a great concept or good intention, it is a real technology that is already deployed and being used as you are reading this book. Several hardware manufacturers already have Java-powered devices on the market, and even more wireless software companies have released applications that can be run directly on a wireless phone or pager.

This is a great time to jump into wireless Java development. The technologies have all come together, and the marketplace is craving anything and everything wireless. Given its background as a programming language for networked devices, Java is the ideal technology to take wireless mobile computing to the next level. This book is designed to not only explain how wireless Java fits into the big scheme of computing and the Internet, but also to get you started with loads of sample code that you can use to spearhead your own wireless development efforts.

In summary, the wireless revolution is still young, and Java is positioned to play a major role in its success. This book will get you right into the thick of things and show you how to build powerful and practical Java applications for wireless mobile devices.

How This Book is Structured

As the title suggests, this book is organized into 21 lessons that are intended to be read a day at a time. There are no penalties if you take more than a day to finish a given lesson, and no monetary rewards if you speed through them faster! The lessons themselves are organized into five parts, each of which tackles a different aspect of wireless Java development.

In Part I, "Getting Started with J2ME and MIDP," you are introduced to the J2ME technology and how it addresses wireless mobile devices:

- On Day 1, "Java 2 Micro Edition: The Big Picture," you see the "big picture" of J2ME by learning about the relationship between J2ME and other wireless technologies. You also find out about how J2ME is organized into configurations and profiles, as well as the hardware and software requirements of Java-powered wireless mobile devices.
- On Day 2, "Assembling a J2ME Development Kit," you assemble a J2ME development kit by installing Sun's J2ME Wireless Toolkit. You also learn about several of the popular visual development environments that directly support J2ME development.
- On Day 3, "Getting to Know the J2ME Emulator," you learn that it isn't necessary to have a physical mobile device in order to build and test J2ME applications (MIDlets). More specifically, you learn how to use the J2ME emulator to emulate MIDlets on your desktop computer.
- On Day 4, "Building Your First MIDlet," you are officially introduced to MIDlets, which are the J2ME equivalents of Java applets. In fact, you build your first MIDlet in this lesson, and then test it within the J2ME emulator.
- On Day 5, "Working within the CLDC and MIDP APIs," you spend some time getting acquainted with the two APIs that are used to construct MIDlets: the CLDC

and MIDP APIs. You find out where these APIs overlap the standard Java 2 APIs and where they branch out on their own. This lesson gives you a good feel for the extent of the J2ME technology in terms of features.

- Day 6, "Creating Custom Device Profiles," wraps up this part of the book with an interesting twist by showing you how to create and use custom device profiles within the Motorola J2ME emulator. A custom device profile effectively lets you create any device as a test suite for MIDlets. This lesson shows you how to create a hypothetical Nintendo GameBoy device profile.

Part II, "Digging Deeper into MIDlet Development," lives up to its name by taking you several steps deeper into MIDlet development. More specifically, the following topics are addressed in this part of the book:

- Day 7, "Building Graphical MIDlets," starts this part of the book off with a bang by guiding you through the graphical portion of the MIDP API, which is used to draw graphics in MIDlets. Along with learning a great deal about MIDP graphics, you also develop a pretty interesting slide show MIDlet.
- On Day 8, "Making the Most of MIDlet GUIs," you go on a whirlwind tour of MIDP graphical user interfaces (GUIs), and the various classes and interfaces that are involved in creating them. This is one of the most hands-on lessons because GUIs play such an important role in virtually all MIDlets. In this lesson you develop a mortgage calculator MIDlet that you can show off to your real estate agent friends.
- On Day 9, "Exploring MIDlet I/O and Networking," you are introduced to networking as it applies to wireless mobile devices. Along with learning a great deal about the MIDP classes and interfaces that facilitate networking and I/O, you also develop a MIDlet that retrieves famous quotes (fortunes) from a remote server.
- Day 10, "Being Your Own Wireless Meteorologist," represents a shift in the book toward extreme practicality by guiding you through the development of a MIDlet that retrieves live weather conditions for a user-specified city. This weather MIDlet is an excellent example of how a MIDlet can utilize the Internet for live data retrieval.
- Day 11, "Wireless Navigation," continues in the spirit of Day 10 by stepping you through the creation of a MIDlet that provides detailed driving directions between two user-specified cities. You enter two cities and the MIDlet queries the Internet for driving directions and then displays them. You'll never get lost again with this MIDlet!
- Day 12, "Optimizing MIDlet Code," completes Part II by explaining some of the limitations of MIDP devices and how they impact the performance of MIDlets. The

end result is that you learn several techniques for squeezing the most performance out of your MIDlets.

Part III, “Wireless Information Management with MIDlets,” addresses the management of information on mobile devices. You develop several interesting data-centric MIDlets as you work through the following topics:

- Day 13, “Using the MIDP Record Management System (RMS),” introduces you to the MIDP Record Management System (RMS), which provides a simple means of storing persistent MIDlet data. Because MIDP programming typically doesn’t involve direct file access, you’ll find RMS an extremely handy feature of the MIDP API.
- On Day 14, “Staying in Touch with Your Contacts,” you put your newfound RMS knowledge to good use by building a contact manager MIDlet that enables you to keep track of personal and business contacts. This MIDlet operates as somewhat of a mobile electronic Rolodex, something you might find useful when you’re away from your home or office.
- On Day 15, “Managing Your Finances,” you build a personal finance MIDlet that enables you to enter financial transactions. This MIDlet functions somewhat as an electronic check register, which is something many of us could use to help keep from losing receipts while on the go.
- Day 16, “Bidding on the Go,” continues with another financial MIDlet by leading through the development of an auction MIDlet. With this MIDlet you can keep track of auctions from the convenience of your mobile phone or pager.

Part IV, “Entertainment without the Wires,” takes on a slightly lighter note than Part III by exploring MIDlet animation and games:

- Day 17, “Creating Animated MIDlets,” introduces sprite animation and how it is implemented using the MIDP API. Sprite animation forms the basis for many video games, so you will probably find the code in this lesson to be highly reusable in your own MIDlets.
- On Day 18, “An Ode to Pong: Creating MIDlet Games,” the sprite code from Day 17 is put to use in a working game that is roughly similar to the classic *Frogger* game. This game serves as a great example of how the different disciplines of MIDP programming come together in an interesting MIDlet.
- Day 19, “The Next Level of MIDlet Gaming,” takes you a step further into the realm of gaming by leading you through the design and development of a strategy game including a computer opponent that relies on artificial intelligence to attempt

to outwit you. You will no doubt find this game invaluable if you have any interest in mobile game development.

Part V, "Exploring J2ME Wireless Technologies," wraps up the book by exploring the present and future of J2ME devices and related technologies:

- Day 20, "A J2ME Wireless Tour," provides a tour of the current crop of wireless mobile J2ME devices and applications. These are the devices that you'll likely be targeting with your MIDlet development, and the applications that you might be interested in trying out as a mobile device user.
- Day 21, "The Future of J2ME," peers into the crystal ball and assesses the role of J2ME in other types of devices and future mobile technologies. This lesson is intended to help you think about the future and how it will impact J2ME and wireless mobile devices.

What You'll Need

This book assumes that you have a solid understanding of the Java programming language and have hopefully developed a few applets or standalone applications in Java. You aren't expected to be a Java expert, but the book does assume a basic level of Java understanding. It might help if you've used a mobile phone or pager before, just so you understand the size and makeup of mobile devices. Beyond that, you aren't really required to know anything about wireless mobile devices or J2ME. To compile and test the sample code presented throughout the book you will need a computer running a platform for which Sun's J2ME Wireless Toolkit is available, which as of this writing is Windows. You'll also need the latest standard Java 2 SDK in order to compile and package MIDlet code. Optionally, you are free to use a visual development environment that supports J2ME development. Day 2 provides an overview of some of these development environments.

On the CD-ROM

This book is a very hands-on programming book, and as such it contains a considerable amount of sample code. All of this code is located on the accompanying CD-ROM for your convenience. I encourage you to use this code as a starting point for your own MIDlets. In addition to the source code for the sample MIDlets covered throughout the book, the CD-ROM also includes compiled and packaged versions of the MIDlets that you can immediately try out. This enables you to quickly get started learning the benefits of wireless J2ME development. Have fun!

PART I

Getting Started with J2ME and MIDP

1 Java 2 Micro Edition: The Big Picture

2 Assembling a J2ME Development Kit

3 Getting to Know the J2ME Emulator

4 Building Your First MIDlet

5 Working within the CLDC and MIDP APIs

6 Creating Custom Device Profiles

DAY 1

Java 2 Micro Edition: The Big Picture

At one time Java was simply a programming language with a slant toward the Internet. Although that meaning certainly still holds true, it's easy to get lost in the many Java technologies that have arisen from the humble beginnings of a simple net-centric programming language. Java has grown by leaps and bounds and has reached far beyond its initial purpose of enabling network devices. It is interesting—and not necessarily coincidental—that Java is now returning to its early roots with J2ME, which stands for Java 2 Micro Edition. Because this book is essentially a guide to using J2ME to construct wireless applications, it makes sense to begin with a clear explanation of what J2ME is and is not. Please understand that this book assumes a fundamental knowledge of Java, so this first lesson jumps directly into J2ME and how it applies Java to wireless devices.

Today you explore the architecture of J2ME and how it relates to the other two main Java technologies, J2SE (Standard Edition) and J2EE (Enterprise Edition). You also learn about how J2ME is organized, along with the specific types of mobile devices targeted by it. More specifically, the following topics are covered in this introduction to J2ME:

- Assessing the role of Java in the wireless world
- Understanding J2ME and how it relates to other Java technologies
- Examining the architecture of J2ME through configurations and profiles
- Learning the two most important wireless J2ME acronyms: CLDC and MIDP

Java Without Wires

Before computers invaded Hollywood and led us to take for granted the ingenuity and craftiness that went into special effects, I can remember watching movies and trying to figure out how certain effects were carried out. One such effect was an object floating in the air, which was often created by gluing a wire to the object and suspending it from above. Although this effect might have fooled less sophisticated audiences in the early days of special effects, it wouldn't fly at all now. What does this have to do with Java and wireless computing? In the very near future the concept of a wire tethered to a mobile computer will seem just as "hokey" as the floating "flying saucers" in an Ed Wood movie.

The concept of a truly wireless mobile computer is new to the vast majority of computer users. We might have already become accustomed to wireless mobile phones, but the idea of being able to connect to the Internet in a similar fashion is something with ramifications we've yet to fully realize. The vast majority of us are still accustomed to plugging a modem into a phone jack to connect to the Internet remotely. It's pretty obvious that being able to check e-mail and surf the Web is very beneficial in a wireless scenario. Whether you're a business traveler connecting to a corporate intranet while waiting in an airport terminal or a restless author of a computer book looking for deals on eBay while in a dentist's waiting room, connecting to the Internet wirelessly is a process that we could all use in one way or another. However, e-mail and Web clients are only the beginning. Even with their limited processing power and small screens, it is now possible to develop interactive software for mobile devices that takes advantage of the wireless network connection.

Not surprisingly, Java is the technology that is making it possible to build custom, interactive software for wireless mobile devices. This is interesting because it brings Java full circle to its original purpose. As I mentioned at the opening of this chapter, Java originally began as a programming language that enabled networked devices to communicate with each other. More specifically, Java started out as a project at Sun with people studying how to put computers into everyday household items. One primary focus of the project was to have these computerized devices communicate with each other. As it played out, Sun was a little ahead of its time in applying Java to network everyday devices. However, the company was quick to react and ended up making Java a huge success by targeting it for the Web.

Now that technology and public perception have caught up with Java's earlier aspirations, Sun has gone back to the drawing board and retooled Java for the mobile computing domain. Not only is a version of Java now designed for the constraints of mobile devices, but this version is also well suited for wireless networking. This version of Java is known as J2ME, which stands for Java 2 Micro Edition. You learn more details about how J2ME fits into the existing Java framework in the next section titled "What is J2ME?" For now, let's continue a little further into the discussion of how Java fits into the wireless world.

You might wonder what exactly constitutes a wireless mobile device. Existing devices that are either already wireless or about to acquire wireless functionality include mobile phones, pagers, personal digital assistants (PDAs), and pocket computers. Although it isn't necessarily new for devices such as mobile phones and pagers to include wireless network functionality, it is very significant that many of them are now supporting Java. By saying that a wireless device supports Java, I mean the device includes a Java virtual machine that is capable of running Java programs. As you learn later in this chapter, these Java programs are a little different from traditional Java applets and applications in that they are designed to run within the limited hardware and software configurations of mobile devices. Figure 1.1 shows the Motorola i2000 mobile phone with built-in Java support.

FIGURE 1.1

The Motorola i2000 mobile phone includes a Java virtual machine that is capable of running Java programs.

You might be interested in knowing some vendors in addition to Motorola that have endorsed Java in their mobile device solutions and are rolling out products and services to support the J2ME platform. The following are some of the major vendors with wireless Java products and services in the works:

- Motorola
- Nokia
- Sony
- Mitsubishi
- Siemens
- Research In Motion (RIM)
- Sprint PCS
- NTT DoCoMo
- LG Telecom
- NEC
- Matsushita/Panasonic
- Fujitsu
- Symbian
- SmarTone
- Far EasTone
- Telefonica
- One 2 One

With this list of vendors signed on, it's apparent that Java will likely change the face of mobile communications dramatically.

It is important to note that the general notion of a wireless device is changing as technology enables wireless functionality to be broadly applied to new applications. For example, home appliances potentially present an interesting application of wireless technology because in some situations it could be beneficial for appliances to communicate with one another. As an example, the oven in my house gives off a lot of heat and causes the kitchen to warm up noticeably at times. If my heating system could communicate with the oven and know when the oven is on, it could temporarily override the thermostat for the entire house and shut down so that the kitchen doesn't get too hot. Granted, this isn't exactly a life-altering application of wireless technology, but it does demonstrate how communication among traditional household devices could sometimes prove beneficial. Of course, the logic that carries out the communication would be driven by Java.

The point of this discussion isn't to sell you on the benefits of wireless computing or give you predictions about wireless devices to come. The point is that Java is positioned to be the enabling technology for such devices, and will quite likely enjoy a second surge in popularity as wireless devices begin to perform more tasks than merely facilitating communication. Information is infinitely more valuable when you can interact with it, and Java makes interactive information access possible on devices ranging from mobile phones to pagers to maybe, some day, even your coffee pot.

What is J2ME?

If you've ever used Java to create basic applets or applications, you probably used the standard edition of Java. In its latest release, this version of Java is known as J2SE, or Java 2 Standard Edition. J2SE is the core set of tools and APIs used to construct Java applets and applications, and is being used widely to construct both Web-based applets and standalone applications. At one time only one version of Java existed, and although it had no designation such as "standard edition," for all intents and purposes that was exactly what it was. It wasn't until Java 2 that Sun realized it might make sense to broaden the scope of Java for enterprise computing.

Ahh, enterprise computing, a world where twenty computers are better than ten, and two hundred computers are often not enough. Enterprise computing differs from traditional computing in that a single application is typically spread across a distributed network of computers and accessed remotely. In enterprise computing, common business logic is shared across one or more applications and/or business units. Additionally, an enterprise application usually involves the aggregation of data from many disparate computing environments. The Web itself is somewhat of an enterprise computing system, but traditional enterprises differ because they typically involve databases and many business rules about how data is accessed, managed, and maintained. If you aren't into enterprise computing you'll be glad to know that it doesn't necessarily play a direct role in wireless Java programming. So why do I mention it? The Java world has been buzzing as of late over J2EE, a version of Java designed solely for building enterprise applications in Java. J2EE stands for Java 2 Enterprise Edition, and is different from J2SE because it adds significant functionality to support enterprise applications.

Why not just include enterprise support in J2SE and keep things simple? The answer to this question has to do with efficiency. Enterprise applications require some heavy-duty networking, I/O, and database features that simply aren't applicable to other types of applications. Because these features incur a significant amount of memory and storage overhead, not to mention additional API complexities, Sun thought it best to break out J2EE as a separate product. It is worth noting that J2EE is a superset of J2SE, which means that J2EE contains all of the functionality of J2SE with the addition of the enterprise computing support.

Now that you understand about J2SE and J2EE, let's turn our attention to J2ME, which as you know stands for Java 2 Micro Edition. Any guesses regarding how it might differ from J2SE and J2EE? Given that wireless mobile devices have less computing power and smaller screens than their desktop counterparts, it stands to reason that J2ME represents a simplified version of J2SE with a reduced feature set. J2ME is actually a subset of J2SE that supports a minimal set of features that are applicable to mobile devices, both wireless and wired. To better understand the relationship of these supersets and subsets, look at Figure 1.2.

FIGURE 1.2

The relationship among J2ME, J2SE, and J2EE.

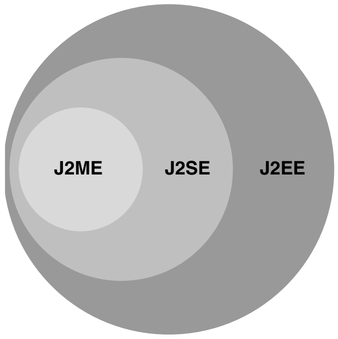


Figure 1.2 shows how J2ME is a subset of J2SE, which is itself a subset of J2EE. In practical terms, this means that J2SE is a simplified version of J2EE without enterprise computing support. Similarly, J2ME is a simplified version of J2SE with functionality that targets mobile devices. Together, these three Java development suites (J2ME, J2SE, and J2EE) comprise the Java 2 technology.

J2ME is admittedly a little different from J2SE and J2EE in that it is designed to vary somewhat according to the specific type of device being targeted. In other words, the J2ME APIs aren't etched in stone. A portion of J2ME is fixed and applies to all devices, while another portion is defined specifically for a certain kind of device such as a mobile phone or a PDA. Later in the section titled "Configurations and the CLDC," you learn more about how the J2ME API is organized.

J2ME and Other Wireless Technologies

Now that you have an idea of what J2ME is, I'd like to clarify what it isn't. The idea of wireless networking on a mobile device is not all that new, although J2ME represents a new twist on the concept. Several other wireless technologies are already in existence. It is important to understand what J2ME has to offer with respect to these technologies, and what the future holds in terms of whether J2ME will supplant or peacefully co-exist with them. Perhaps the three most important wireless technologies at present are WAP, SMS, and Bluetooth. The next few sections explore the relationship between J2ME and these other wireless technologies.

J2ME and WAP

You have probably heard the expression “wireless Web.” If so, you were quite likely hearing someone talk about WAP, even if they didn’t realize it. WAP, which stands for *Wireless Application Protocol* and is pronounced so that it rhymes with “slap,” is a technology that enables wireless devices to receive data from the Internet and display it on their constrained displays. You can think of WAP as essentially a technology that supports a minimal Web browser on wireless devices. However, as its name clearly states, WAP is not an application but a protocol.

The only drawback to WAP is that it must be supported on both the client (device) and the Web server, and a WAP Gateway, an intermediary between the Internet and the device’s mobile network, must also exist. The WAP Gateway is responsible for converting WAP requests into traditional Web requests, and vice versa. Web pages for WAP are somewhat different from traditional Web pages in that they are written in a special markup language called WML (Wireless Markup Language), as opposed to the familiar HTML. WML also supports a scripting language called WMLScript, a simplified version of JavaScript.

WAP is already in use and is supported on a variety of mobile phones. The question you are then most likely pondering is how does WAP relate to J2ME? Some people have wrongly compared the two technologies as if they are competitors. J2ME is no more a competitor of WAP than Java is a competitor of HTML. J2ME provides a means of creating client applications that access and manipulate data through a wireless network, typically the Internet. WAP, on the other hand, is simply a protocol for browsing the Web on a mobile device. It is perfectly possible and quite likely that J2ME will peacefully co-exist with WAP.

J2ME and SMS

If you’ve ever used a pager to send text messages back and forth to colleagues in a boring meeting, you are no doubt familiar with SMS, which stands for Short Messaging Service. SMS is a relatively simple technology that supports sending and receiving short text messages on mobile devices such as mobile phones and pagers. Another interesting feature of SMS is its support for unified messaging, which allows you to access voice mail, e-mail, and faxes from a mobile device.

Given the specialized messaging nature of SMS, it's pretty obvious that there is no direct competition between it and J2ME. Granted, if you're trying to develop a bare-bones chat or mail client using J2ME then you might see some overlap with SMS, but otherwise there is no problem with a device supporting both SMS and J2ME.

J2ME and Bluetooth

By far the most hyped of the three wireless technologies we're discussing is Bluetooth, which is the much anticipated wireless networking technology that aims to change the way we all work and play. Okay, maybe it won't change things for all of us, but the potential possibilities of Bluetooth are pretty exciting. The idea behind Bluetooth is to enable short-range wireless communications between devices. Practically speaking, Bluetooth aims to replace most of the cables used to connect computing devices with a wireless radio connection. This means that you no longer have to worry about tripping over the printer cable that crosses in front of the doorway to your study. With Bluetooth, it might finally be possible to eliminate the rat's nest of cables that inevitably pile up behind most computers. Perhaps more significant is the freedom it will allow in being able to roam around your office or house without having to worry about plugging and unplugging devices.

What's the connection between Bluetooth and J2ME? Well, there is no direct connection because Bluetooth tackles a hardware networking issue whereas J2ME is clearly on the software side of things. Bluetooth does factor into wireless mobile devices, but not in a way that benefits J2ME at the moment. When Bluetooth is first introduced into mobile phones, it will likely be to enable a phone to share a network connection with another computing device such as a laptop. On the other hand, there is no reason why Bluetooth couldn't be used to facilitate a wireless network connection between a mobile device and an office LAN, for example. The bottom line is that it still isn't clear how Bluetooth will impact J2ME, but it's safe to say that these technologies will likely benefit each other at some point in the near future.

Configurations and the CLDC

Along with introducing a different perspective for developing Java applications, J2ME introduces several new terms that are essential in understanding the J2ME architecture. The first of these new terms is a *configuration*, which is a minimum set of APIs that is useful for developing applications to run on a range of devices. Configurations are very important because they describe the core functionality required of a range of devices. A standard configuration for wireless devices is known as the *Connected Limited Device Configuration*, or CLDC. The CLDC describes a minimum level of functionality required for all wireless mobile devices. The CLDC takes into consideration factors such as the amount of memory available to such devices along with their processing power.

New Term *configuration*—A minimum set of APIs that is useful for developing applications to run on a range of devices.

I mentioned earlier that J2ME is unlike J2SE and J2EE because its API isn't etched in stone. The CLDC is part of what facilitates this extensible API. More specifically, it is important to understand that the CLDC describes an API for a certain family of devices. Although this device family is relatively broad, it is fully expected that additional configurations will eventually describe other families of devices that might or might not be wireless or mobile. You can think of the CLDC as the set of absolutely essential classes and interfaces that are necessary for building applications for wireless mobile devices.

Note The CLDC is currently the only configuration defined for J2ME. However, it is expected that additional configurations will be developed as Sun targets other types of devices for J2ME development.

To be a little more specific, the CLDC clearly outlines the following pieces of information with respect to wireless mobile devices:

- The subset of Java programming language features
- The subset of functionality of the Java virtual machine
- The core APIs required for wireless mobile application development
- The hardware requirements of the wireless mobile devices targeted by the CLDC

You might assume that the entire Java programming language is available for use in mobile devices, but in fact a few features are disabled under the CLDC due to the limited processing power of mobile devices. These core language issues impact the Java virtual machine, resulting in a smaller feature set for the virtual machine used in CLDC devices. The feature set of the virtual machine is certainly important, but perhaps the most significant information outlined by the CLDC is the set of core APIs that must be supported by a mobile device implementation of J2ME.

The final piece of the CLDC puzzle is the actual hardware requirements of a wireless mobile device. These requirements are specified as minimums, meaning that a device must have at least the following hardware attributes to qualify as a CLDC device:

- 160KB of total memory available for Java
- 16-bit processor
- Low power consumption (often battery power)
- Network connectivity (often wireless with a 9600 bps or less bandwidth)

Given this hardware description, the kinds of devices that are targeted by the CLDC become clearer. CLDC devices include, but are not limited to, mobile phones, pagers, PDAs, pocket computers, and home appliances. It is fully expected that other useful devices might eventually fall within the CLDC realm; just keep in mind that they must adhere to the hardware requirements.

Examining these requirements a little closer, it is important to understand exactly what is meant by the minimum memory requirement of 160KB. The CLDC goes a step further by clarifying the manner in which memory is used within the 160KB:

- 128KB of non-volatile memory for the Java virtual machine and CLDC API libraries
- 32KB of volatile memory for the Java runtime system

The CLDC and the Java Programming Language

I mentioned earlier that the CLDC imposes a few limitations on the Java programming language. The primary limitation is a lack of floating-point math support, which means that you can’t perform any floating-point math operations in an application developed for use with the CLDC. The reason for the lack of floating-point support is that CLDC devices typically don’t have the hardware to carry out such operations. This means that the operations would have to be carried out in software, which is not really a good idea considering the performance hit that an application would suffer. The end result is that you can’t use floating-point date types, literals, or operations in CLDC applications.

Another limitation in the Java language is the removal of the `Object.finalize()` method. The `finalize()` method is called whenever an object is removed from memory to help clean up any related resources. The CLDC doesn’t require the Java virtual machine to support object finalization; therefore the `finalize()` method is not available for use in CLDC applications.

Exception handling is the last area of the Java programming language that is limited in some way by the CLDC. More specifically, the CLDC takes somewhat of a delegated approach to exception handling because many error conditions in mobile devices are device-specific. In other words, it is difficult to define standard error classes that apply to all possible CLDC devices. The solution is to support a limited set of error classes and then allow specific errors and exceptions to be handled by a device-specific API.

The CLDC and Security

As with the standard and enterprise editions of Java, the mobile version of Java also takes security quite seriously. However, given the minimalist approach taken by the CLDC, it stands to reason that security is implemented on a small budget. If you've spent much time developing applets, you're probably familiar with the sandbox security model. This model stipulates that an applet can only perform certain operations that fit within a safe set of API features (the sandbox). Any features outside of these safe features aren't allowed, and are therefore considered to be located outside of the sandbox. So, an applet is not allowed to play outside of its sandbox. If you didn't spend much time in a sandbox as a kid, just understand that the sandbox security model essentially means that some features are off-limits and cannot be accessed.

Deciding what API features are off-limits is where things get tricky. The idea is to allow developers to build rich, powerful applications without hindering them too much with security constraints. On the other hand, any computing device connected to a network that is capable of running dynamically downloaded applications is at a serious security risk. So, a fine line must be drawn that serves both purposes. Following are the basic guidelines of the CLDC sandbox security model:

- Java class files must be verified as valid Java applications
- Only predefined CLDC Java APIs are available for use
- No user-definable class loaders are allowed
- Only native features exposed by the CLDC can be accessed; no additional native code can be used

Notice that the security restrictions imposed by the CLDC are relatively low-level restrictions because the CLDC is really a core specification. It is expected that an additional layer, which is defined in a device profile, will impose further restrictions. The next section clarifies what exactly a profile is, and the specific profile that targets wireless mobile devices.

The MID Profile (MIDP)

You now have a pretty good feel for how a configuration fits into the J2ME architecture. On top of a configuration sits a *profile*, which is a more specific set of APIs that further targets a particular type of device. A configuration describes in general terms a family of devices, whereas a profile gets more specific and isolates a particular type of device within that family. Practically speaking, a profile simply adds additional API functionality that pertains to the device with which it is associated.

NEW TERM *profile*—A specific set of APIs that targets a particular type of device.

As you learned in the preceding section, the CLDC is the first J2ME configuration, and describes a family of mobile devices that supports certain core Java features. The Mobile Information Device Profile, or MIDP, is a profile built on top of the CLDC that describes a wireless mobile device such as a mobile phone or pager. The MIDP specification describes a MIDP device as "a small, resource-constrained, wireless-connected mobile information device." Because the MIDP is built on top of the CLDC, the CLDC APIs are available to developers building MIDP applications.

In addition to specifying APIs for use in MIDP application development, the MIDP also describes minimum hardware and software requirements for a MIDP device. The next two sections outline these hardware and software requirements. Keep in mind that the MIDP requirements are applied in addition to the CLDC guidelines you learned about earlier today. In other words, the MIDP takes the CLDC device requirements and nails them down a bit further.

Hardware

The hardware requirements for MIDP devices are an important part of the MIDP standard. These requirements are broken down into the following device properties:

- Memory
- Input
- Display
- Networking

Memory Requirements

The memory requirements for MIDP devices are as follows:

- 128KB of non-volatile memory for the MIDP API libraries
- 32KB of volatile memory for the Java runtime system
- 8KB of non-volatile memory for persistent application data

If you recall from the earlier memory requirements for the CLDC, this MIDP memory description differs only in the addition of the 8KB required for persistent application data.

Input Requirements

The input requirements of MIDP devices stipulate that a MIDP device must have a keyboard or touch screen. The keyboard can be either one-handed or two-handed, and it is possible that a device might have both a keyboard and a touch screen. Notice that a mouse isn't an input requirement because it is unlikely that a mobile device would be capable of using a mouse. However, it is quite possible for a device to use a stylus with a touch screen.

Display Requirements

The display requirements for MIDP devices are a little more interesting because the screens for mobile devices represent one of the most constrained hardware properties. MIDP device displays must have a screen size of 96 pixels by 54 pixels with a 1-bit color depth. This means that the screen must be at least 96 pixels wide and 54 pixels high, and must be at least black and white. Furthermore, the aspect ratio of the display must be 1:1, which simply means that pixels must be square. Believe it or not, many computer monitors don't have a 1:1 aspect ratio, which means that pixels are actually rectangular in shape. Not so with MIDP device screens!

Networking Requirements

The last hardware area targeted by the MIDP specification is networking, which dictates the minimum networking support required of a MIDP device. A MIDP device must have a two-way, wireless network connection of some sort. It is expected that such a connection may be intermittent, such as a dial-up connection, and that it also may have limited bandwidth (9600bps). This is important because it informs developers that they must be very conscious of bandwidth issues when designing applications that transfer information across a network connection to and from a MIDP device. Of course, the up side to this requirement is that all MIDP devices are required to have a network connection of some sort, which means that you can always assume some degree of network access when designing MIDP applications.

Software

Although much has been said about the battle over operating systems in the desktop computer market, the reality is that there are only a few big players in the desktop OS fight. You can step up to virtually any desktop computer and be relatively assured that you'll immediately recognize the operating system. Strangely enough, the same cannot be said of mobile devices. This is primarily because operating systems simply aren't as critical to mobile devices. Granted, every device must have an operating system to work properly, but it isn't as important to lock into an operating system for compatibility purposes. This means that you can't assume very much about the operating system of a mobile device.

Of course, the cross-platform nature of Java helps alleviate concerns over the wide range of mobile device operating systems. Even so, the MIDP specification lays some ground rules about what is expected of the operating system in a MIDP device. Following are the major software requirements for MIDP devices:

- A minimal kernel to manage low-level hardware features such as interrupts, exceptions, and scheduling
- A mechanism to read from and write to non-volatile (persistent) memory
- A timing mechanism for establishing timers and adding time stamps to persistent data
- Read/write access to the device's wireless network connection
- A mechanism to capture user input from a keyboard or touch screen
- Minimal support for bitmapped graphics
- A mechanism for managing the life cycle of an application

These requirements, although somewhat minimal, still provide a reasonably rich set of features that are available for use by a MIDP application. Keep in mind that specific APIs are set forth by MIDP, which you learn about in Day 5, "Working with the CLDC and MIDP APIs." For now I just want you to become familiar with the general software guidelines set forth by the MIDP specification.

Summary

Today you were introduced to Java 2 Micro Edition, which is also referred to as J2ME. In addition to learning about the relationship between J2ME, J2SE, and J2EE, you learned about the kinds of devices for which you can develop software with J2ME. You then explored the architecture of J2ME and how it is broken down into configurations and profiles. Two very important acronyms (CLDC and MIDP) were explained, along with their roles in J2ME application development. Admittedly, today's lesson only laid the ground rules for J2ME. This was intentional, however, because the remainder of the book is spent digging into the J2ME technology and making it do things. Now that you have a solid understanding of J2ME at a high level, you're ready to move on and begin learning about J2ME development tools.

In the next lesson you learn about the different tools and utilities that comprise a J2ME development environment. Along with installing the core J2ME tool suite, you also learn about some of the third party visual development environments that support J2ME development.

Q&A

Q Why is it necessary for J2ME to be divided into configurations and profiles?

A The idea behind configurations and profiles is to make J2ME as flexible as possible. Unlike desktop computers, which are all more or less the same in terms of fundamental capabilities, small computing devices can vary a great deal when it comes to their capabilities. Not only can screen sizes vary greatly, but also some types of devices might not even have screens. For this reason, the J2ME API needs to be extremely flexible. By tying the J2ME API to configurations and profiles, Sun allows the API to acquire or lose features depending upon the specific type of computing device. The CLDC and MIDP describe a set of J2ME APIs applicable to wireless mobile devices.

Q What devices are currently available for CLDC and MIDP development?

A As of this writing, there is a J2ME virtual machine and runtime system available for mobile phones, pagers, PDAs, and pocket computers. Motorola has been noticeably involved in J2ME and offers several Java-powered phones that utilize J2ME. J2ME is also supported in Palm-compatible devices. To find out the latest releases of J2ME available, visit the J2ME Web site at `http://java.sun.com/j2me/`.

Q Do I need to own a J2ME device in order to begin developing J2ME applications?

A Absolutely not. Most J2ME development tools include an emulator that allows you to run J2ME applications on your desktop computer.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What three editions of Java are currently available?
2. What is a configuration?
3. What is a profile, and how does it relate to a configuration?
4. What does the acronym MIDP stand for?

Exercises

1. Think about how you would use a wireless mobile device if it could run virtually any networked application. Make a list of some of the applications you'd like to find, use, and possibly even develop for such devices.
2. Use the Web to find a few different Java-enabled devices that support the CLDC and MIDP specifications. Analyze the hardware details of these devices and compare them to the requirements laid out by the CLDC and MIDP specifications.

DAY 2

Assembling a J2ME Development Kit

As with traditional Java development, J2ME programs require various development tools for compilation and testing. However, J2ME applications have unique requirements that make the development process a little different from traditional J2SE or J2EE applets and standalone applications. More specifically, J2ME classes must go through a pre-verification process that verifies that the code doesn't violate J2ME security constraints. J2ME applications also require a special emulator to test them in an environment similar to a real J2ME device. Of course, you could always test an application directly on a physical device, but it is much more efficient to do the majority of testing with an emulator and then save the final testing for the physical device. The standard Java compiler is still used to build J2ME applications, after which the additional tools enter the picture.

Today you will explore the development tools required for constructing J2ME applications. You are introduced to the basic command-line tools required for J2ME application development, as well as some visual tools that are readily available. Following are the major topics covered in this lesson:

- Understanding the basics of J2ME application development
- Getting acquainted with the J2ME Wireless Toolkit
- Assessing the available J2ME visual development tools
- Understanding the role of Web servers in J2ME applications

A J2ME Development Primer

The remainder of the book is devoted to the intricacies of developing J2ME applications. Although this lesson doesn't directly delve into application development, it does require you to have a basic understanding of how J2ME applications are constructed to appreciate the significance of J2ME development tools. So, let's start by referring to J2ME applications by their proper name; J2ME applications developed to the MIDP specification are referred to as *MIDlets*. More specifically, a MIDlet only utilizes classes and interfaces defined in the CLDC and MIDP APIs, which you learn about on Day 5, "Working with the CLDC and MIDP APIs." You can think of a MIDlet as an applet designed specifically for use on mobile devices.

NEW TERM *MIDlet*—A Java application developed to the MIDP specification for mobile devices.

In addition to being constrained to the CLDC and MIDP APIs, MIDlets must also go through a special pre-verification process. Like applets and applications, MIDlet classes are stored in Java bytecode files with a `.class` file extension. However, MIDlet classes must be verified prior to distribution to guarantee that they don't perform any illegal operations. The reason for this pre-verification has to do with the limitations of the virtual machine used in mobile devices. This virtual machine is referred to as the *K Virtual Machine*, or *KVM* for short. To keep the KVM as small and efficient as possible, it is desirable to minimize the amount of verification it must perform on a MIDlet class. So, some of this verification is handled at design-time during the pre-verification process.

Note

The naming of the KVM has to do with the resource requirements of the virtual machine. It basically means that the KVM requires Ks of memory, as opposed to MBs. In other words, the KVM is designed to fit within several kilobytes (K) of memory, as opposed to a full-blown J2SE virtual machine that might take up several megabytes of memory (MB).

Pre-verification takes place just after compilation, and results in a new class file being generated that is verified and ready for testing or distribution. This brings us to another distinguishing factor between MIDlets and applets/applications: MIDlets must be specially packaged in JAR files for distribution. You might think that because applets are also packaged in JAR files this is nothing new. Although this is true, JAR files are optional for applets but are strictly required for MIDlets. MIDlets also require that several extra fields be included in the manifest file for the JAR file, as well as an entirely new descriptor file known as a JAD file. Following are the pieces of information typically included in a MIDlet JAR file:

- MIDlet classes
- Support classes
- Resources (images, sounds, and so on)
- Manifest file (`.mf` file)
- Application descriptor (`.jad` file)

Note

In case you aren't experienced with JAR files, they are used to package Java classes into compressed archives for more efficient distribution. Prior to J2ME, JAR files were used to package applets for improved organization and reduced download time. A manifest file is a special text file included in a JAR file that describes the classes contained within the archive.

The JAD file I mentioned is an application descriptor file that provides additional information about the MIDlets contained within a JAR file. Notice that I said MIDlets, plural. Yes, multiple MIDlets are typically contained within a single JAR file. Such a collection of MIDlets is referred to as a *MIDlet Suite*. The idea behind a MIDlet Suite is that it enables multiple MIDlets to share the limited resources available in a mobile device. The KVM for the device is required to provide a means of navigating and selecting a particular MIDlet to run from the MIDlet Suite. The JAD file included in the JAR file for a MIDlet Suite is instrumental in providing information relevant to installing and accessing the individual MIDlets.

NEW TERM *MIDlet Suite*—A collection of MIDlets packaged together in a single JAR file.

You thought this section was about J2ME development, not the inner workings of JAR files and MIDlet Suites. The reality is that you need to know a little about MIDlets to understand the process of developing them. The MIDlet development process is summarized in the following steps:

1. Edit
2. Compile
3. Pre-verify
4. Emulate
5. Test on device

Steps 1 and 2 are hopefully pretty familiar to you because they parallel traditional Java applet and application development. Steps 3 and 4 are where things look a little different. You already know that a pre-verification step is required to ensure that a MIDlet isn't trying to do something that isn't allowed under the MIDP specification such as trample system memory. Step 4 is where you test a MIDlet using a special tool called an *emulator*. A J2ME emulator does exactly what its name implies—it emulates a physical mobile device on your desktop computer. This allows you to test MIDlets within the comforts of your desktop computer without having to download the code onto a device. Debugging is also made much simpler when a MIDlet is running within an emulator.

New Term *emulator*—A software tool that emulates a physical mobile device on a desktop computer.

Note

In some ways you can think of the Java SDK applet viewer as an emulator for applets because it imitates the Java functionality of a Web browser. In other words, the applet viewer is to applets what a J2ME emulator is to MIDlets. Technically speaking, this comparison isn't entirely accurate because emulators are designed to emulate hardware. Nonetheless, the comparison makes sense to the extent that the applet viewer provides a testing environment for applets in the same way that a J2ME emulator provides a testing environment for MIDlets.

The final step in the MIDlet development process is testing the MIDlet on a physical device. In reality, you will probably test your MIDlets on several different devices to make sure that they perform as expected on each. This testing phase typically takes place after you've ironed out the bugs in a MIDlet, and you're very close to shipping it as a final product. In other words, the bulk of the MIDlet testing takes place in the J2ME emulator prior to entering the physical device into the picture.

Now that you have an idea of how a MIDlet is developed, I'd like to shift gears and take a look at the major development toolkits associated with MIDlet development. Following are the three major toolkits that you will use to design and construct MIDlets:

- Java 2 SDK
- J2ME Wireless Toolkit
- Visual development environment

The Java 2 SDK is the familiar standby that you have probably already used extensively to construct applets and standalone applications. The J2ME Wireless Toolkit is a standard toolkit made available by Sun that serves as an add-on to be used in conjunction with the Java 2 SDK. The J2ME Wireless Toolkit includes a bytecode verifier and several J2ME emulators for verifying and testing MIDlets. In addition to the standard J2ME Wireless Toolkit, several mobile device manufacturers offer their own toolkits that include additional tools for MIDlet development. For example, Motorola's SDK for J2ME provides a configuration editor that enables you to create custom device profiles. (For more information on Motorola J2ME development programs, visit `http://www.motorola.com/idendev/`.) Using the configuration editor, you can test MIDlets on virtually any custom MIDP device.

The last toolkit mentioned is a visual development environment that you can use in lieu of the command-line tools made available in the Java 2 SDK and J2ME Wireless Toolkit. Visual development environments automate many menial tasks that go into developing a MIDlet, and can make the development process much smoother. You learn about several J2ME visual development environments later in the lesson in the section titled "Visual J2ME Development Environments."

Inside the J2ME Wireless Toolkit

The J2ME Wireless Toolkit is a set of development tools made available by Sun that facilitates MIDlet development when used with the Java 2 SDK. You can download the J2ME Wireless Toolkit from `http://java.sun.com/products/j2mewtoolkit/`. The toolkit essentially boils down to the following two command-line tools:

- Bytecode verifier
- J2ME emulator

Note

The J2ME Wireless Toolkit also includes a visual development environment called KToolBar, which you learn about later in the section titled "Visual J2ME Development Environments."

After compiling MIDlet files using the standard Java compiler, the resulting class files must be verified using the bytecode verifier. The J2ME emulator provides a test environment for running MIDlets without having to use an actual mobile device. Although these two tools form the basis for J2ME development using the J2ME Wireless Toolkit, an additional tool, the *configuration editor*, is made available by some device vendors. The configuration editor enables you to create custom device profiles. Although the configuration editor doesn't ship with Sun's J2ME Wireless Toolkit, it is readily available in Motorola's SDK for J2ME. The next few sections explore the major tools found in these development environments in more detail.

Note

If you plan on using Sun's Forte for Java visual development environment to develop MIDlets, then you must install it *prior* to installing the J2ME Wireless Toolkit. During installation, the J2ME Wireless Toolkit will look to see whether Forte for Java has been installed, in which case it will ask if you want it to be integrated with Forte for Java. If you intend to use Forte for Java for MIDlet development, you must select "Integrated" during the installation of the J2ME Wireless Toolkit. Otherwise you can just select "Stand Alone." You learn about Forte for Java later in this lesson in the section titled "Visual J2ME Development Environments."

The Bytecode Verifier

According to the CLDC specification, the Java virtual machine used in a mobile device must be able to detect and reject invalid class files. The virtual machine employed in J2SE and J2EE includes verification functionality to carry out this exact detection. However, the overhead required to perform such detection on mobile devices with limited memory and processing power proves to be too burdensome. For this reason, the burden of verifying class files in J2ME is removed from the virtual machine (KVM). The verification is handled instead on a desktop PC or server as part of a pre-verification process that is carried out before a class file is downloaded to a device. During the development phase of a MIDlet, the pre-verification of class files is performed using a special tool called the *bytecode verifier*, which ships standard with the J2ME Wireless Toolkit.

Note

If you're curious about how a class file is verified in J2ME, I'll clue you in that a special attribute is assigned to each method in a class file; the bytecode verifier tool adds these attributes if a class verifies OK. When the class is loaded into the KVM, the attribute is checked to make sure the class has been verified as safe for use, thereby relieving the KVM of having to perform the verification itself. As you might have guessed, verified class files are slightly larger than unverified files because of the addition of the attributes.

The bytecode verifier tool is named pre-verify, and is executed at the command-line similarly to the standard Java 2 SDK tools. The preverify tool has the following syntax:

```
preverify -classpath CLASSPATH -d DEST_DIR SRC_DIR
```

The path to any support classes (including MIDP classes) is specified in the `CLASSPATH` argument. The `DEST_DIR` and `SRC_DIR` arguments specify the destination directory where the new verified classes are to be written and the directory where the source class files are stored. Because you don't actually use the bytecode verifier until Day 4, "Building Your First MIDlet," it isn't necessary for you to memorize the use of the bytecode verifier. Nonetheless, I wanted to get you acquainted with the tool because it is a major part of your J2ME toolset.

It is worth noting at this point that if you elect to use a visual development environment to build and test MIDlets, you will likely forego using the command-line bytecode verifier directly because most visual development environments are designed to automatically invoke the tool behind the scenes. You learn about some of these visual tools later today in the section titled "Visual J2ME Development Environments."

The J2ME Emulator

Perhaps even more important than the bytecode verifier is the J2ME emulator, which is a tool that plays an invaluable role in testing MIDlets during development. The reason for using an emulator has to do with the practical difficulties of downloading code onto a physical device over and over as you diagnose bugs and make changes. It is much more effective and efficient to test MIDlet code in a desktop environment, and only resort to testing on a physical device in the latter stages of MIDlet development.

The J2ME Wireless Toolkit includes a handy command-line emulator that is executed using the `java` interpreter. This emulator is very flexible and can emulate a wide range of J2ME devices including mobile phones and pagers. An image of the device is actually displayed on the desktop screen in a special window while the MIDlet executes within the virtual LCD screen (see Figure 2.1). You can use the mouse to interact with the device by clicking buttons on the image.

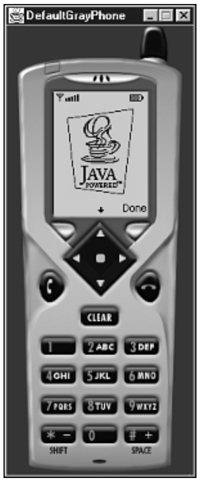


FIGURE 2.1
The J2ME emulator displays an image of a mobile device with a virtual LCD screen in which a MIDlet executes.

The execution of the emulator using the `java` interpreter is a little trickier than the bytecode verifier, primarily because the emulator supports several properties that can be used to configure it. Because tomorrow's lesson focuses entirely on using the emulator, I'll spare you the details for right now. Keep in mind that similar to the bytecode verifier, the emulator is automatically invoked by most visual development environments when it comes time to test a MIDlet.

The Configuration Editor

Last on the list of J2ME tools is the configuration editor, which isn't part of the standard J2ME Wireless Toolkit. Instead, you will find the configuration editor as part of toolkits offered by device manufacturers such as Motorola. The Motorola SDK for J2ME includes the configuration editor, which allows you to create and edit device profiles. This means that you have the freedom to dictate specifics regarding devices that are emulated such as the device image, screen size, and available buttons, to name a few. The Motorola configuration editor includes profiles for several Motorola phones such as the iDEN i1000 and the StarTac.

Note

The J2ME Wireless Toolkit also enables you to create and modify device configurations, but it doesn't provide any special tools to help you along in the process. However, as you learn on Day 6, "Creating Custom Device Profiles," it isn't too terribly difficult to edit these configurations by hand.

The configuration editor is a very interesting and powerful tool that gives you a great deal of control over the emulation environment used for MIDlet testing. You learn how to create and edit device profiles using the configuration editor on Day 6.

On the subject of the Motorola SDK for J2ME, it's worth mentioning that this SDK can be used in place of the J2ME Wireless Toolkit. In other words, the Motorola SDK for J2ME includes its own verification tool and emulator in addition to the configuration editor. Although the Motorola verification tool and emulator are different from the Sun versions that are included with the J2ME Wireless Toolkit, they perform the exact same function. The decision is ultimately yours as to which tool suite you use, but if you know that you are going to initially target Motorola wireless devices then it's probably a good idea to start with the Motorola SDK for J2ME because it includes profiles for Motorola devices.

Visual J2ME Development Environments

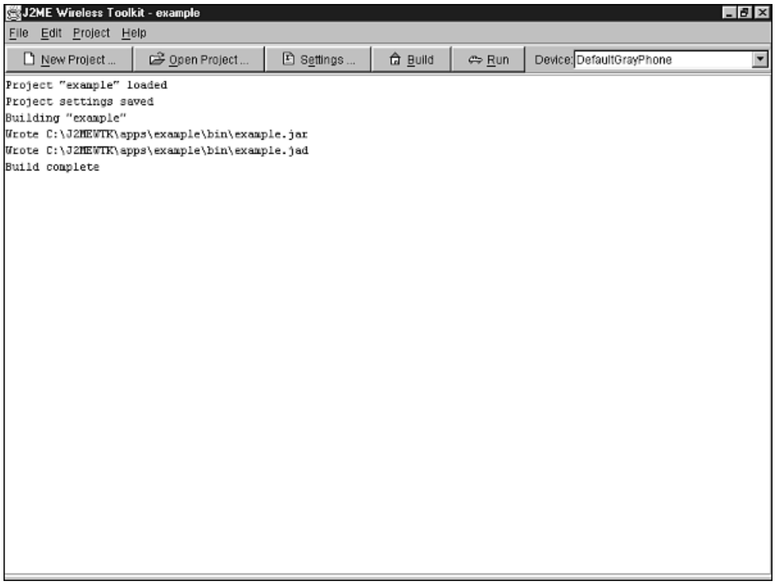
Throughout this lesson I've alluded several times to visual development environments and how they improve and enhance the process of designing and building MIDlets. Both the standard J2ME Wireless Toolkit and the Motorola SDK for J2ME provide a simple command-line environment for constructing MIDlets, but a visual development environment can speed things up in terms of compiling, verifying, and testing MIDlets. Because J2ME is still a relatively new technology, not as many visual environments are available for it as are available for J2SE or J2EE development. Nevertheless, you still have several options. The following are the main visual development environments that directly support J2ME as of this writing:

- KToolBar
- Forte for Java (with J2ME Module)
- CodeWarrior for Java
- JBuilder Handheld Express

The next few sections explore each of these development environments and what they have to offer.

KToolBar

KToolBar is by far the simplest of the visual development environments that currently support MIDlet development with J2ME. In fact, it doesn't even include an editor for editing source code. Instead, KToolBar focuses on the task of managing source code files and automating the build process. Using the KToolBar application, you can forego the command-line J2ME tools and perform the compile, verification, and emulation steps from a single environment. Figure 2.2 shows the KToolBar application with an open J2ME project.



```
Project "example" loaded
Project settings saved
Building "example"
Wrote C:\J2MEWTK\apps\example\bin\example.jar
Wrote C:\J2MEWTK\apps\example\bin\example.jad
Build complete
```

FIGURE 2.2
KToolBar provides a minimalist approach to visual J2ME development.

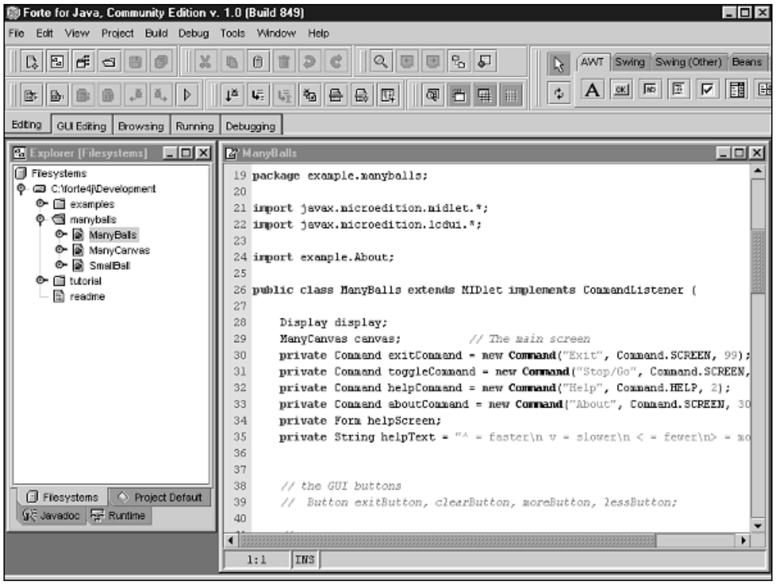
Although the KToolBar tool is admittedly a minimalist visual environment, the upside is that it is included free with the J2ME Wireless Toolkit. So, whether or not you decide to use it, it is automatically installed when you install the standard J2ME wireless tools. Just keep in mind that if you should decide to use the KToolBar for J2ME development, you'll need to find a suitable text editor for editing the source code files. If you already have a visual Java development environment, even if it doesn't yet support J2ME, you might find it useful for editing J2ME source code files.

Forte for Java

Forte for Java is Sun's comprehensive development environment for J2SE and J2EE development. Forte for Java doesn't come standard with support for J2ME; however, a special add-on module that adds J2ME support to Forte is included in the J2ME Wireless Toolkit. The end result is a very powerful visual J2ME development environment that significantly improves the MIDlet development process. The Forte environment includes the following major components that facilitate visual MIDlet development:

- GUI text editor
- Class browser
- Web browser

The GUI text editor in Forte for Java enables you to edit Java source code with color context highlighting. This makes it much easier to read and understand code because keywords and other language constructs are colored uniquely. The class browser provides a graphical view of the classes that comprise a MIDlet project. Using the class browser, you can easily navigate the methods and fields within a class, as well as analyze the compiled classes stored in a JAR file. The Web browser built into Forte for Java is convenient because it allows you to view J2ME's HTML-based documentation directly in the Forte environment. Figure 2.3 shows the Forte for Java development environment.


Forte for Java, Community Edition v. 1.0 (Build 849)
File Edit View Project Build Debug Tools Window Help
AWT Swing Swing (Other) Beans
Editing GUI Editing Browsing Running Debugging
Explorer [Filesystems]
Filesystems
C:\forte4j\Development
examples
manyballs
ManyBalls
ManyCanvas
SmallBall
tutorial
readme
Filesystems Project Default
Javadoc Runtime
ManyBalls
1:1 INS

```
package example.manyballs;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

import example.About;

public class ManyBalls extends MIDlet implements CommandListener {

    Display display;
    ManyCanvas canvas;          // The main screen
    private Command exitCommand = new Command("Exit", Command.SCREEN, 99);
    private Command toggleCommand = new Command("Stop/Go", Command.SCREEN,
    private Command helpCommand = new Command("Help", Command.HELP, 2);
    private Command aboutCommand = new Command("About", Command.SCREEN, 30
    private Form helpScreen;
    private String helpText = "^ = faster\n v = slower\n < = fewer\n> = mo


    // the GUI buttons
    //  Button exitButton, clearButton, moreButton, lessButton;

```

FIGURE 2.3

Forte for Java offers a high-powered visual approach to building MIDlets.

One interesting thing about Forte for Java is that it is built using an open, modular architecture that enables it to be extremely extensible. That's one of the reasons why it's possible to extend the base tool with J2ME support with ease. The Community Edition of Forte for Java is available on the accompanying CD-ROM. You can also find more information at the Forte for Java Web site, which is located at `http://www.sun.com/forte/ffj/`.

CodeWarrior for Java

Similar to Forte for Java, Metrowerks' CodeWarrior for Java is a full-featured visual development environment that offers support for J2ME development. CodeWarrior for Java includes familiar graphical development features such as a GUI text editor and class browser. You can visit the CodeWarrior Web site at `http://www.codewarrior.com/` to learn more.

JBuilder Handheld Express

The last of the visual development environments with J2ME support is JBuilder Handheld Express. JBuilder is the popular Java development environment that originally built on the success of Delphi, a Pascal-based development environment by Inprise. JBuilder offers many of the same features as Forte and CodeWarrior, and with the release of JBuilder Handheld Express support is now available for J2ME development. As of this writing, JBuilder Handheld Express targets Palm devices only, but I expect to see support for a wider range of wireless devices such as phones and pagers very soon. Stop by the JBuilder Handheld Express Web site at `http://www.inprise.com/jbuilder/hhe/` for more information.

J2ME Web Servers

You might not think of a Web server as a traditional Java development tool, but if you are building a wireless application that transmits and receives data to and from a data source of some sort, a Web server will likely play a role in the overall system design. Similar to a Web-based application that runs as a suite of Java applets, a wireless suite of MIDlets also uses the Web as a means of accessing and sharing data. Such a wireless application must rely heavily on a Web server with support for application data access. Many existing Web servers can fit the bill for such an application, but the following are two popular open-source Web servers that are well suited for wireless systems:

- Apache
- Enhydra

You are probably already familiar with Apache, which is the most widely used open-source Web server in existence as of this writing. Apache is developed by The Apache Software Foundation, and is available on the accompanying CD-ROM. You can also visit the Apache Web site at `http://www.apache.org/` for more information.

Whereas Apache is the leading open-source Web browser, Enhydra is perhaps the leading Java/XML application server. Enhydra plays a similar role as Apache, but its focus is centered more on e-commerce applications that require a full-blown application server. Lutris Technologies, which still oversees the Enhydra open source project, initially created Enhydra. Enhydra is available on the accompanying CD-ROM, or you can visit the Enhydra Web site at `http://www.enhydra.org/`.

Summary

This lesson introduced you to J2ME development tools and why they are important to the J2ME development process. The most important tool suite for J2ME development is the J2ME Wireless Toolkit, which is the official toolkit made available by Sun. Other vendors such as Motorola also have similar toolkits that are tailored more to their specific wireless products. If you want to simplify things a little and avoid the hassles of working with command-line tools, you can step up to a visual development environment such as KToolBar, Forte for Java, CodeWarrior for Java, or JBuilder Handheld Express, all of which were covered in today's lesson. Finally, you learned in the lesson how Web servers play an important role in many J2ME wireless applications.

In exploring the tools that are used to build J2ME applications, you also learned a fair amount about J2ME development in general. This information will serve you well as the rubber hits the road in the next lesson when you start running MIDlets with the J2ME emulator.

Q&A

Q Why are MIDlets packaged together in a MIDlet suite?

A The idea behind the MIDlet suite is essentially to conserve resources. All the MIDlets in a MIDlet suite are loaded together, and therefore share memory and other device resources. Unlike applications that execute in a desktop environment, each MIDlet in a MIDlet suite functions as a part of a singular application in terms of device resources. So, if you load a MIDlet suite containing several games, for example, each game executes independently but they all share the same memory and processor resources of the device.

Q Do I need to install both the J2ME Wireless Toolkit and the Motorola SDK for J2ME?

A Not necessarily. Both tool suites provide a bytecode verifier and a J2ME emulator, so either of them will allow you to build and test MIDlets. However, only the Motorola SDK for J2ME includes a configuration editor for creating custom device profiles. For this reason, I encourage you to install the Motorola SDK for J2ME. On the other hand, the J2ME Wireless Toolkit includes valuable documentation and example code that you will probably find useful. So, you might want to also install it so that you have access to those resources.

Q If I decide to use a visual development environment to create MIDlets, do I still need to install a command-line tool suite (J2ME Wireless Toolkit or the Motorola SDK for J2ME)?

A Yes. Currently, the visual development environments for J2ME utilize the command-line tools behind the scenes to carry out the pre-verification process and to emulate MIDlets, so it is still necessary to install a command-line tool suite.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and start you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What must happen to a compiled MIDlet class before it can be executed in a wireless environment?
2. What is the virtual machine in mobile devices called?
3. What are the two primary development tools included in the J2ME Wireless Toolkit?
4. What is the purpose of the configuration editor that is included in the Motorola SDK for J2ME?

Exercises

1. Install the Motorola SDK for J2ME and the J2ME Wireless Toolkit. You definitely need to install the Motorola SDK for J2ME because it includes the configuration editor tool that is used on Day 6. It is available at `http://www.motorola.com/indendev`

2. Using the accompanying CD-ROM, install the visual development environment, Forte for Java, and take it for a test drive. Keep in mind that the KToolBar development environment is installed automatically with the J2ME Wireless Toolkit.

DAY 3

Getting to Know the J2ME Emulator

Unlike traditional Java applets and applications, which can be executed using a Web browser or Java interpreter during the development phase, MIDlets must be executed using a special emulator that emulates a physical mobile device. A J2ME emulator is an important tool that ships with both the standard J2ME Wireless Toolkit and with other similar toolkits that are made available by device vendors such as Motorola. Although it is certainly possible to test a MIDlet directly on a mobile device, an emulator streamlines the process and enables you to work entirely on a desktop computer throughout the development process. It is still important to test a MIDlet on a physical device as it nears completion, but the emulator plays a critical role in testing a MIDlet prior to that point.

This lesson introduces you to the J2ME emulator that ships with the J2ME Wireless Toolkit, as well as the emulator that ships with the Motorola SDK for

J2ME. Both emulators serve the exact same function, but running each of them is a little different. You also learn today how to launch an emulator from within a visual development environment. These are the major topics you cover in today's lesson:

- Understanding the strengths and weaknesses of the J2ME emulator
- Becoming acquainted with the types of devices targeted by the emulator
- Running the emulator from the command line
- Running the emulator using a visual development environment

J2ME Emulator Essentials

Today you start to see first-hand why J2ME is such an interesting technology. The J2ME emulator is the tool that enables you to run MIDlets on a desktop PC and simulate how the MIDlet will run on a physical device. Before getting into the specifics of how to run MIDlets within the emulator, I'd like to quickly go over some of the benefits of a J2ME emulator, along with a few of its limitations. Following are three primary benefits of using a J2ME emulator:

1. You can delay testing on a physical device until the final stage of MIDlet development.
2. You can test a MIDlet on a variety of different target devices, including custom devices.
3. You can track specific aspects of a MIDlet's execution such as class loading, method calls, and garbage collection.

The first benefit is pretty obvious and was mentioned in the previous lesson. It reflects the fact that the emulator serves as a great stand-in for a physical device during the early and middle stages of MIDlet development; you'll still want to run a MIDlet through its paces on a physical device late in the development process.

The second benefit alludes to the fact that the emulator operates with respect to a device profile. You learned about device profiles in Day 1, "Java 2 Micro Edition: The Big Picture," but to quickly recap, a device profile models the properties of a physical device. The emulator is very flexible in that it enables you to test a MIDlet on multiple device profiles, which effectively tests the MIDlet on multiple devices. You can even define custom devices (through custom profiles) and test MIDlets on them; you learn how to do this on Day 6, "Creating Custom Device Profiles."

The last benefit listed has to do with the emulator's capability to provide diagnostic information about a MIDlet as it is executing. You learn more about the specific kinds of diagnostic information made available by the emulator a little later in this lesson.

Before you begin to think that the J2ME emulator is the most amazing development tool ever envisioned, I need to temper your excitement by highlighting a few of its limitations. None of these limitations are killers, but it is important to understand where the emulator falls short of a physical device in terms of testing MIDlets. Then you can focus on these aspects of a MIDlet when you do test it on a physical device. Following are the aspects of a MIDlet that cannot be controlled or tested by the emulator:

- Execution speed
- Available memory
- Application manager

Physical mobile devices vary widely in their hardware, which means that processor speed is most definitely a variable that cannot be nailed down by a MIDlet developer unless you target a very specific device and ignore others. Even though you can't necessarily make assumptions about the speed of the processor in a device, it would be nice to be able to test a MIDlet at a variety of different speeds to see how it responds. Unfortunately, the J2ME emulator doesn't currently factor device speed into the emulation process. So you'll have to resort to testing a MIDlet on physical devices when it comes to assessing the speed of execution across different processors.

The processor speed will likely vary considerably across different mobile devices, and the memory available to a MIDlet will quite likely vary as well. Because available memory can dramatically impact the execution of a MIDlet, it is important to test a MIDlet within the memory constraints of each target device. Unfortunately, the J2ME emulator doesn't currently enable you to vary the memory available to a MIDlet, so it doesn't help much in terms of assessing the impact of available memory on a MIDlet. Chalk up memory as another aspect of your MIDlet that you'll need to test directly on a physical device, along with the speed of execution.

The last limitation of the J2ME emulator is less critical than the first two, and has to do with the application manager that is used on mobile devices to manage MIDlets. The application manager on a device is responsible for allowing you to install, remove, and execute individual MIDlets. Currently no support is available for application management in the emulator, which means that you can't completely test the management of MIDlets on a device using the emulator. However, this is a task that is easy enough to test on a physical device when your MIDlet is ready for deployment.

Beyond these easily identifiable weaknesses in the J2ME emulator, you also must consider the fact that any emulator is merely approximating a physical device. This means that an emulator will inevitably yield slightly different results from a real device, which is why you can't rely 100% on emulators for all J2ME software testing. So, although emulators serve a great role in enabling you to accomplish a great deal of developmental testing on a PC, you must always plan on spending ample time testing on a physical device.

Now that I've reversed myself and convinced you that a J2ME emulator isn't all that great, let me reiterate that it is still an extremely valuable tool outside of the few limitations I just mentioned. With that in mind, let's move on and address the relationship between the emulator and physical devices.

Emulating Physical Devices

The J2ME Wireless Toolkit is designed as an all-purpose J2ME development tool suite that is geared toward a wide range of wireless mobile devices. For this reason, you won't find any mention of a specific manufacturer's name or device model in the J2ME Wireless Toolkit. Instead, general device types are supported. More specifically, the J2ME Wireless Toolkit includes device profiles for the following devices:

- Pager
- Minimum phone
- Default gray phone
- Default color phone

As you might have guessed, these devices are listed in order of increasing functionality. In other words, the pager and minimum phone device profiles represent devices with less functionality than the default color phone; the default color phone device profile has a larger screen that includes colors. Even so, not all physical devices will have such features, so it can be valuable to test a MIDlet on all of these device profiles. Table 3.1 lists the specific attributes of these device profiles.

TABLE 3.1 Attributes for the Device Profiles Supported in the J2ME Wireless Toolkit

Device Profile	*Screen Size*	*Screen Colors*	*Key Set*	*# Soft Buttons*
Pager	150×60	black/white	QWERTY	0
Minimum phone	96×54	black/white	ITU-T	0
Default gray phone	96×128	256 grayscale	ITU-T	2
Default color phone	96×128	256 colors	ITU-T	2

Table 3.1 reveals how much variance is possible in J2ME devices. Notice that the pager has a screen that is horizontally oriented, whereas the phones have vertical screens. Also, the pager and minimum phone have only black and white displays, the gray phone supports 256 shades of gray, and the color phone supports 256 colors. The key sets on the devices vary as well. The pager includes a full QWERTY key set similar to your computer keyboard—this corresponds to the popular alphanumeric pagers such as those produced by Research In Motion (RIM). The ITU-T key set is based upon the familiar keypad commonly found on most mobile phones in use today. Finally, the soft buttons are special buttons that are accessible from MIDlets to perform application functions.

Note

ITU stands for International Telecommunication Union, and is the United Nations specialized agency dealing with telecommunications. The ITU is heavily involved in developing and overseeing telecommunications standards.

Although the general devices in the J2ME Wireless Toolkit are valuable for testing MIDlets without regard for specific brands and models of devices, you might want to specifically target a popular phone or pager for testing. In this case, you'll need to obtain the profile for the device. One way to accomplish this is to use the J2ME toolkit made available by the manufacturer of the device in question. As an example, the Motorola SDK for J2ME includes profiles for the following devices:

- Generic phone
- Motorola iDEN
- Motorola iDEN i1000
- Motorola StarTac

Aside from the generic phone, the device profiles included in the Motorola toolkit correspond to real products that Motorola has on the market. By testing a MIDlet on these devices within the J2ME emulator, you can more accurately approximate how the MIDlet will function on the real device, particularly how it will look on the device display.

Running the Emulator from the Command Line

The J2ME emulator included with the J2ME Wireless Toolkit is a command-line tool, which means that it must be run from the command line unless you are using a visual development environment. Even if you are using a visual development environment, it's not a bad idea to learn how to use the command-line emulator because you are likely to become more acquainted with the different emulator properties by running directly from the command line. Keep in mind that most visual J2ME development environments employ the same command-line emulator; they just launch it automatically without you ever having to deal with the command line yourself.

The standard J2ME emulator ships as part of the J2ME Wireless Toolkit. In addition to this emulator, some device vendors make available their own emulators for use in testing MIDlets with their respective devices. Motorola includes such an emulator as part of their Motorola SDK for J2ME. The next two sections show you how to run MIDlets using the emulators in the J2ME Wireless Toolkit and the Motorola SDK for J2ME.

The J2ME Wireless Toolkit

Unlike many command-line Java tools, the J2ME emulator is not an executable native application that you can run directly in your native operating environment. The emulator is actually a Java application, which means that you must use the Java interpreter to run it. Also, keep in mind that J2ME is built on a set of special APIs that must somehow be available to the Java interpreter to successfully emulate a MIDlet. Associating the APIs with the emulator is as simple as altering the `CLASSPATH` environment variable before invoking the Java interpreter.

The `run.bat` Script

To help make the process of setting the class path and invoking the emulator a little easier, the J2ME Wireless Toolkit includes a special script file called `run.bat` that sets the `CLASSPATH` variable accordingly and launches the emulator in the Java interpreter. This script is located in the `apps\example\bin` directory within the J2ME Wireless Toolkit installation directory. Figure 3.1 shows the result of executing the `run.bat` script at the command-line.

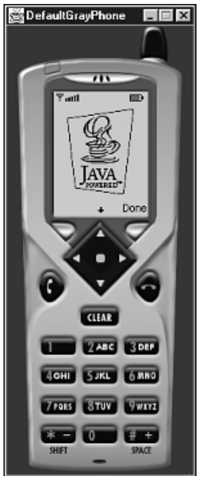
Note The `run.bat` script can be named differently on platforms other than Windows. It is common for Windows scripts (batch files) to be named with a `.bat` file extension.

As Figure 3.1 shows, the `run.bat` script launches the J2ME emulator and displays a splash screen showing the Java logo. Clicking the upper-right button with the mouse removes the splash screen and makes available several MIDlets for testing. In a moment I'll get into the details of these MIDlets and how to interact with them using the emulator's phone interface. For now, I want to take a look inside of the `run.bat` script to see what's going on. Following is excerpted code for the `run.bat` script, which shows how the class path is altered and how the Java interpreter is used to invoke the emulator:

```
cd ..\..\..\

set
CLASSPATH=lib\kvem.jar;lib\kenv.zip;lib\lime.jar;apps\example\bin\example.jar

java -Dkvem.home=. com.sun.kvem.midp.Main DefaultGrayPhone -descriptor apps\
➥example\bin\example.jad

cd apps\example\bin
```

FIGURE 3.1

The `run.bat` *script launches the J2ME emulator, which shows an image of a phone with a Java splash screen.*



The directory changes in the script code are necessary to establish the J2ME Wireless Toolkit installation directory as the root directory for emulation. More important is the modified `CLASSPATH` variable, which is altered to include the paths of relevant J2ME libraries that are required for emulation. The `kvem.jar` and `kenv.zip` files contain classes for the emulator KVM (virtual machine), whereas the lime.jar file contains other J2ME support classes. The `example.jar` file contains the executable code for the various sample MIDlets that are included in the J2ME Wireless Toolkit (more on these later in this section).

With the class path properly set, the remaining step is to launch the emulator using the Java interpreter. The `-D` option to the interpreter is used to set the `kvem.home` system property, which is in turn used by the emulator to identify the root directory of the J2ME Wireless Toolkit. This is the only system property that must be set for the emulator, although there are other emulator properties. The actual emulator class file specified to the Java interpreter is `com.sun.kvem.midp.Main`. The remaining arguments on the command-line are passed along to the emulator. The first emulator argument is the name of the device to emulate, which in this case is `DefaultGrayPhone`. The next argument must be set to `-descriptor`, followed by the name of the application descriptor file (JAD file). The JAD file is responsible for conveying to the emulator how the application is organized.

I mentioned that the `kvem.home` system property is the only required property of the emulator, but that there are several other optional properties. Table 3.2 lists all the properties supported by the J2ME emulator.

TABLE 3.2 Attributes for the Device Profiles Supported in the J2ME Wireless Toolkit

Property	*Type*	*Status*	*Default Value*
`kvem.home`	String	Required	None
`http.proxyHost`	String	Optional	None
`http.proxyPort`	Integer	Optional	None
`kvem.trace.gc`	Boolean	Optional	False
`kvem.trace.calls`	Boolean	Optional	False
`kvem.trace.class`	Boolean	Optional	False
`kvem.trace.exceptions`	Boolean	Optional	False

The `http.proxyHost` and `http.proxyPort` properties are used to set the host and port of the HTTP proxy to be used during emulation. The remaining properties are used to turn on and off tracing information for garbage collection, method calls, class loads, and exceptions. All the tracing properties are set to false by default, which means you must explicitly turn them on to view tracing information during emulation.

The Sample MIDlets

Referring to the emulator session that was invoked by the `run.bat` script file, you probably noticed several MIDlets listed in the phone's display (see Figure 3.2).

All these MIDlets are included as examples in the J2ME Wireless Toolkit, and are available for testing. Following are the sample MIDlets that ship with the standard J2ME Wireless Toolkit:

- **Sokoban**—A game where you slide blocks into slots
- **Tickets**—A concert ticket bidding system

- **Colors**—A demonstration of using colors
- **Stock**—A client/server stock ticker
- **Tiles**—A tile puzzle game
- **Many Balls**—A demonstration of threaded animation
- **Sampler**—A demonstration of several graphics features
- **Properties**—A demonstration of obtaining information about the device
- **Http Test**—A demonstration of establishing an HTTP connection
- **Pong**—A pong game
- **Star Cruiser**—A game where you navigate a ship through a star field
- **Space Invaders**—A space invaders game
- **UIDemo**—A demonstration of various GUI controls

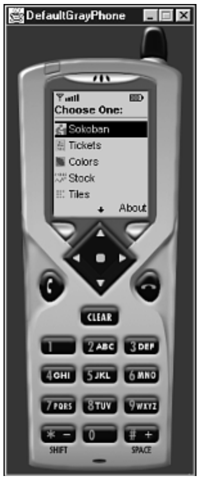


FIGURE 3.2
The `run.bat` *script launches the J2ME emulator with several example MIDlets available for testing.*

All of the MIDlets except for UIDemo are available for testing using the aforementioned `run.bat` script file. UIDemo is itself a MIDlet suite, and includes its own `run.bat` script file that is located in the `apps\UIDemo\bin` directory. Testing the UIDemo MIDlet suite is a great way to become familiarized with the different GUI components that are available for J2ME development. You learn all about these components and how to develop MIDlets using them on Day 8, "Making the Most of MIDlet GUIs."

Running Without a Script

Thus far you've seen how to use a script to invoke the emulator and execute a suite of MIDlets specified in the `examples.jad` file that is included in the J2ME Wireless Toolkit.

Although this is handy for getting started with the emulator quickly, it isn't quite so convenient if you want to run other MIDlets or supply additional properties to the emulator. For this reason, it is important to understand how to invoke the emulator directly from the command line without the help of the standard `run.bat` script file. Of course, the `run.bat` script gives you virtually all the information you need to carry out this task. Following is the general form of invoking the emulator:

```
java -cp ToolkitRoot\lib\kvem.jar;ToolkitRoot\lib\kenv.zip;ToolkitRoot\lib\
➥lime.jar;JARFile
-Dkvem.home=ToolkitRoot [-DProperty=Value] com.sun.kvem.midp.Main
➥DeviceName –descriptor DescriptorFile
```

I realize this looks rather messy, but bear with me because it's really not too bad when you understand what's going on. First of all, let's clarify what the placeholders mean:

- ***`ToolkitRoot`***—The root directory of the J2ME Wireless Toolkit (usually \j2mewtk)
- ***`JARFile`***—The JAR file containing the MIDlet(s) you are testing
- ***`Property/Value`***—A property/value pair for an emulator property (see Table 3.2 for available properties)
- ***`DeviceName`***—The name of the device profile to be used by the emulator
- ***`DescriptorFile`***—The relative or absolute path of the application descriptor (JAD) file

You might have noticed that this command-line form for invoking the emulator is very similar to the `run.bat` script you saw earlier, especially when you consider that the class path settings are now being carried out directly on the command line using the `-cp` option. The real trick is getting the placeholders filled in properly. If you're invoking the emulator from the J2ME root directory, you can just insert a period (`.`) for *`ToolkitRoot`*. Otherwise, you'll need to specify the root directory everywhere *`ToolkitRoot`* appears in the command. The *`JARFile`* specifies the JAR file containing the MIDlet classes that you are testing. The only property you must specify is `kvem.home`, which must be set to the J2ME root directory. The *`DeviceName`* can be set to any of the following standard device profiles: `Pager`, `MinimumPhone`, `DefaultGrayPhone`, or `DefaultColorPhone`. Finally, the JAD file for the MIDlet(s) must be specified as the last argument in the command.

To help you cement in your mind the use of the emulator from the command line, here is an example of invoking the emulator from a single command using the pager profile and the J2ME example MIDlets:

```
java -cp.\lib\kvem.jar;.\lib\kenv.zip;.\lib\lime.jar;.\apps\example\bin\
➥example.jar
-Dkvem.home=. com.sun.kvem.midp.Main Pager -descriptor apps\
➥example\bin\example.jad
```

The result of issuing this command is shown in Figure 3.3, which reveals the flexible nature of device profiles and how much variance there can be among different devices.

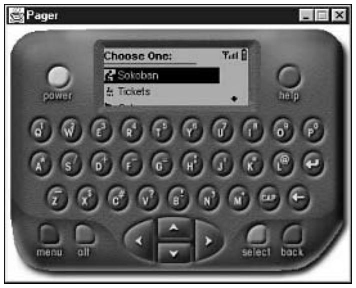


FIGURE 3.3

By specifying Pager as the device name when executing the emulator, a pager device is used as the basis for testing MIDlets.

If you are a command-line guru, you might already realize that this approach has a problem. The problem is that some command lines limit commands to 128 characters, which isn't enough for this hefty command. The workaround is to create your own script (batch) file to house the command. In practice, you will probably want to create scripts for your own MIDlets anyway because typing out such long commands can get cumbersome.

Controlling the Device

Because you can't make any calls or receive pages using the virtual devices represented by the J2ME emulator, it shouldn't be too surprising that you can't touch the screen to click the buttons. Instead, you can use the mouse to click any of the buttons displayed as part of a device in the emulator. Additionally, certain keys on your computer keyboard are mapped to device keys. The numeric keypad on the keyboard maps directly to the keypad on a phone device. Similarly, the letter keys and several command keys on the keyboard are all usable on the pager device. Arrow keys on the keyboard apply to all of the standard devices that are available for testing with the emulator.

The Motorola SDK for J2ME

Now that you know how to run the standard emulator included with the J2ME Wireless Toolkit, it's time to turn your attention to the Motorola SDK for J2ME and its very capable emulator. The drill is very similar to the standard J2ME emulator in that there is a script to help make things easy. The Motorola version of `run.bat` is called `runEmul.bat`, and its excerpted source code follows:

```
@if NOT "%1"=="" set MYCLASS=%1
@if     "%1"=="" set MYCLASS=com.mot.j2me.midlets.bounce.Bounce
```

```
set CLASSPATH=

set J2ME_RESOURCE_DIR=c:\MotoSDK\lib\resources

cd ..\bin

java -Djava.library.path=../lib -classpath ./Emulator.jar;./ConfigTool.jar
➥com.mot.tools.j2me.emulator.Emulator -classpath../demo/midlets;../lib
javax.microedition.midlet.AppManager %MYCLASS% -JSA 1 1

cd ..\scripts
```

One interesting thing about this code is that it accepts a command-line argument, which is referenced by `%1` in the code. The idea is that you can specify the name of a MIDlet class file to `runEmul.bat` and it will run it using the emulator. The second line of the script shows the default behavior if you don't provide a class file—the Bounce example MIDlet is executed.

Instead of using the `CLASSPATH` environment variable, the Motorola emulator specifies dependent libraries directly in the command that invokes the interpreter. One environment variable that it does set, however, is `J2ME_RESOURCE_DIR`, which is the directory where any resources are stored for use with the MIDlets. Instead of using the `CLASSPATH` environment variable, the path of the standard J2ME libraries is stored in the `java.library.path` system property. Additionally, several support JAR files are specified using the `-classpath` option to the Java interpreter.

The emulator class itself is specified as `com.mot.tools.j2me.emulator.Emulator`. Everything in the command that appears after the emulator class is passed along to the emulator application as arguments. So, the first argument to the emulator is the path to the MIDlet(s) being executed. The second argument is the application manager that is responsible for running the MIDlets. The most significant argument is the `%MYCLASS%` argument, which contains the name of the actual MIDlet being executed. Finally, the `-JSA 1 1` arguments at the end of the command are used internally by the emulator and are specified the same for all MIDlets.

Now that you understand how the `runEmul.bat` script works, let's take it for a test drive. If you change to the directory containing the script and simply enter `runEmul` at the command line, the Bounce example MIDlet will be executed, as shown in Figure 3.4.

To execute a different MIDlet, specify the full classname on the command line as the only argument to the `runEmul` script. Following is the command necessary to run the Scribble MIDlet using the Motorola emulator:

```
runEmul com.mot.j2me.midlets.scribble.Scribble
```

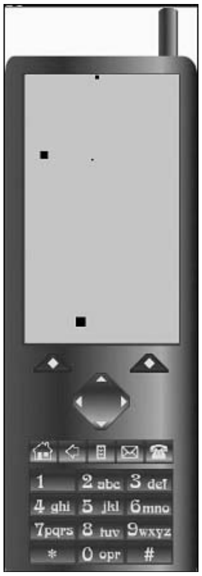


FIGURE 3.4

The `runEmul.bat` *script launches the Motorola J2ME emulator and executes the Bounce sample MIDlet by default.*

The result of this command is shown in Figure 3.5 after I've done some doodling in the Scribble MIDlet.

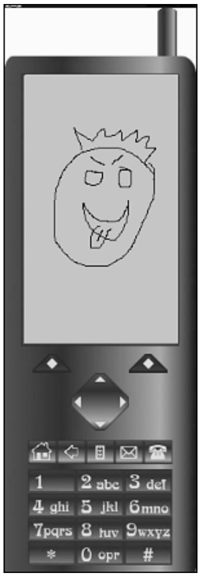


FIGURE 3.5

The Scribble MIDlet is executed in the Motorola emulator by specifying the full classname as the only argument to `runEmul.bat`.

Changing Devices

Although the default device shown in Figure 3.5 is generally reminiscent of a real mobile phone, it would be much more interesting to see an actual Motorola phone in the emulator. If you recall from earlier in the lesson, the Motorola SDK for J2ME supports several commercial Motorola devices, all of which are mobile phones. You can easily tweak the emulator to use one of these devices with a simple property addition to the command that invokes the emulator in the Java interpreter. The property is called `deviceFile`, and is used to specify a device file containing a custom device profile. Following is an example of how the main command in `runEmul.bat` changes to use the Motorola iDEN mobile phone device file:

```
java -Djava.library.path=../lib -classpath ./Emulator.jar;./ConfigTool.jar
➥com.mot.tools.j2me.emulator.Emulator -classpath../demo/midlets;../lib
-deviceFile resources/MotorolaiDENPlatform.props
javax.microedition.midlet.AppManager %MYCLASS% -JSA 1 1
```

Notice in the code that the device file is specified as `resources/MotorolaiDENPlatform.props`. This simple change to the emulator command results in the Motorola iDEN device profile being used instead of the generic phone profile. Keep in mind, however, that you've been using `runEmul.bat` to run MIDlets using the Motorola emulator, as opposed to invoking it directly with the Java interpreter by hand. Fortunately, the Motorola SDK for J2ME includes additional scripts for running MIDlets with each different device:

- **`runEmul.bat`**—Runs a MIDlet using the generic phone.
- **`runMotoiDEN.bat`**—Runs a MIDlet using the Motorola iDEN phone.
- **`runMotoi1000.bat`**—Runs a MIDlet using the Motorola iDEN i1000 phone.
- **`runStarTac.bat`**—Runs a MIDlet using the Motorola StarTac phone.
- **`runMyDevice.bat`**—Runs a MIDlet using a custom device.

All of these scripts except for the last one are used to run the emulator using existing device profiles with which you probably have some degree of familiarity. The last script, `runMyDevice.bat`, is used to run the emulator with an entirely custom device. You learn how to create a custom device and use it with the Motorola emulator on Day 6.

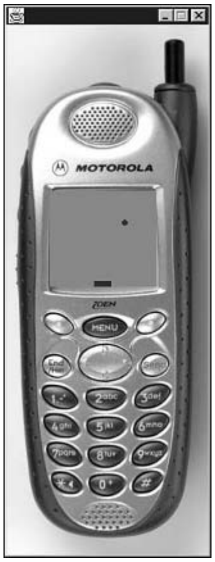
Getting back to the device profiles for Motorola devices, following is an example of executing the PaddleBall example MIDlet in the emulator using the `runMotoiDEN.bat` script:

```
runMotoiDEN com.mot.j2me.midlets.paddleball.PaddleBall
```

Figure 3.6 shows the result of this command, which reveals a photo-realistic Motorola iDEN phone in the emulator.

FIGURE 3.6

The PaddleBall MIDlet is executed in the Motorola emulator using the Motorola iDEN phone.



The Sample MIDlets

Similar to the J2ME Wireless Toolkit, the Motorola SDK for J2ME includes several sample MIDlets that you can try out using the Motorola emulator. Following are the sample MIDlets that ship with the Motorola toolkit:

- **Bounce**—An animation demonstration
- **PaddleBall**—A paddle ball game
- **Scribble**—A demonstration of drawing with basic graphics
- **FontDemo**—A demonstration of using fonts
- **GraphicsDemo**—A demonstration of several graphics features
- **RecordStoreDemo**—A database demonstration
- **AlertTest**—A demonstration of timed and modal alerts
- **ChoiceGroupTest**—A demonstration of the choice group GUI component
- **DateFieldTest**—A demonstration of the date field GUI component
- **FormTest**—A demonstration of forms
- **GaugeTest**—A demonstration of the gauge GUI component
- **KeyEventsTest**—A demonstration of handling key events
- **TextBoxTest**—A demonstration of the text box GUI component
- **TextFieldTest**—A demonstration of the text field GUI component

- **TickerTest**—A demonstration of the ticker GUI component
- **UDP Tutorial MIDlets**—Two MIDlets that demonstrate UDP network communication

All of the MIDlets can be tested using any of the `runXXX.bat` Motorola script files.

Running the Emulator Within a Visual Environment

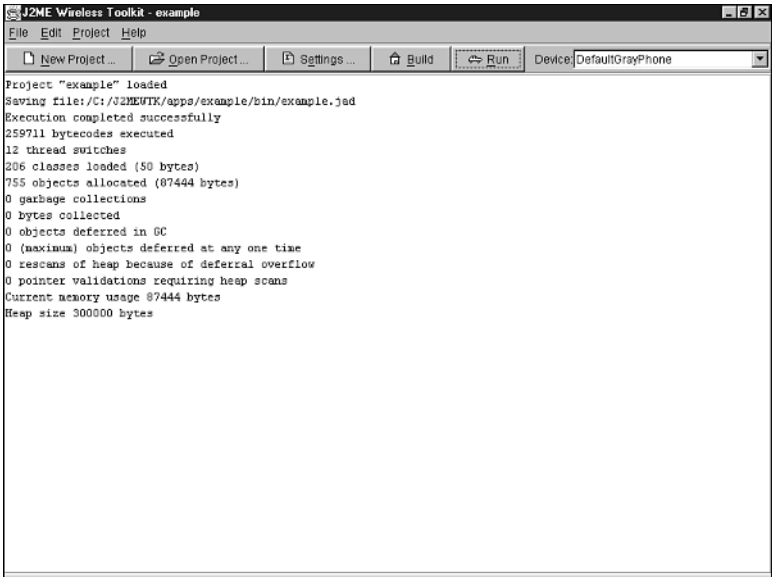
Although it is certainly beneficial and relatively painless to execute the J2ME emulator at the command line, you can opt to use a visual development environment for MIDlet construction, in which case you will probably want to launch the emulator directly from within the visual tool. Fortunately, this functionality is supported in the current crop of J2ME visual development environments. The next couple of sections explain how to utilize the emulator from within the KToolBar and Forte for Java visual development environments.

Note

Both the KToolBar and Forte for Java development environments are designed for use with the standard emulator included with the J2ME Wireless Toolkit, which means that you can't currently use them to invoke the emulator included in the Motorola SDK for J2ME.

KToolBar

The KToolBar development environment makes it very easy to invoke the J2ME emulator on a MIDlet project to test a MIDlet. To use the emulator, you must first open an existing project or create a new one. To open the samle MIDlet suite included with the J2ME Wireless Toolkit, click the Open Project button on the KToolBar main toolbar and then click example in the list of projects. Clicking the Open Project Button in the dialog box selects the sample project and opens it. To execute the MIDlet suite in the J2ME emulator, click the Run button on the toolbar. The J2ME emulator will then be executed just as if you had run it from the command line using the `run.bat` script. In fact, the information typically displayed in the command-line window upon executing the emulator now appears directly in the KToolBar application window. Figure 3.7 shows the output of the emulator as it appears in the KToolBar application window after exiting the emulator.



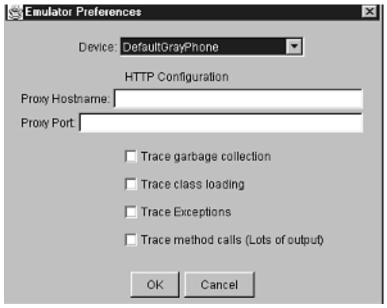
```
Project "example" loaded
Saving file:/C:/J2MEWTK/apps/example/bin/example.jad
Execution completed successfully
259711 bytecodes executed
12 thread switches
206 classes loaded (50 bytes)
755 objects allocated (87444 bytes)
0 garbage collections
0 bytes collected
0 objects deferred in GC
0 (maximum) objects deferred at any one time
0 rescans of heap because of deferral overflow
0 pointer validations requiring heap scans
Current memory usage 87444 bytes
Heap size 300000 bytes
```

FIGURE 3.7
The output that results from running the J2ME emulator appears directly in the KToolbar application window.

Note

Don't forget that you can also compile a MIDlet project by clicking Build on the KToolBar toolbar.

In addition to running the emulator from within the KToolBar application, you can also tweak its properties through a visual interface. To alter these properties, select Preferences from the Edit menu. Figure 3.8 shows the Emulator Preferences dialog box that is displayed after selecting the Preferences command.



Device: DefaultGrayPhone
HTTP Configuration
Proxy Hostname:
Proxy Port:
☐ Trace garbage collection
☐ Trace class loading
☐ Trace Exceptions
☐ Trace method calls (Lots of output)

FIGURE 3.8
The KToolBar application enables you to alter emulator properties through a visual interface.

The most useful property you will likely want to change is the target device, which is easily modified by clicking the drop-down list of available devices. As an alternative to using the Emulator Preferences dialog box, you can also select the target device for the project using the drop-down list that appears on the main toolbar.

Forte for Java

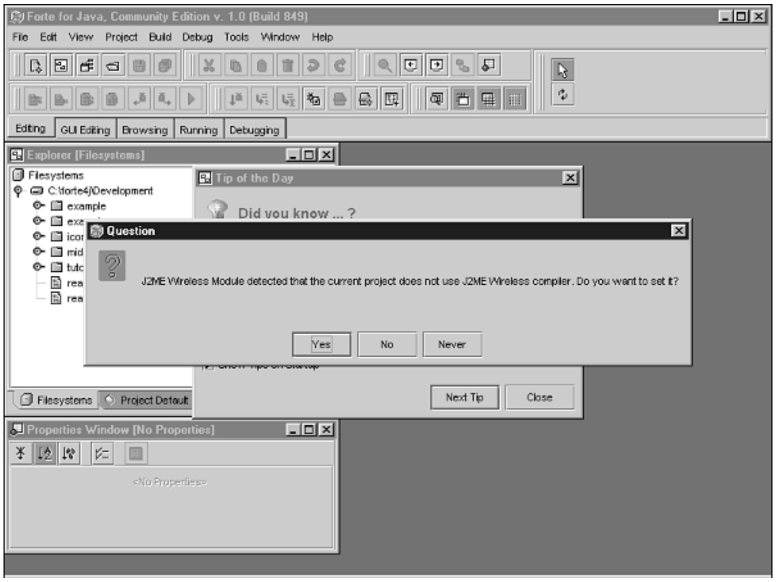
If you're looking for a more comprehensive development environment than KToolBar has to offer, you might be considering Forte for Java. Similar to KToolBar, Forte for Java allows you to test MIDlets by invoking the J2ME emulator from within the visual environment without ever having to enter commands at the command line. To test a MIDlet in the emulator using Forte for Java, you must first open a MIDlet project. Because Forte for Java expects to find all projects in the Development directory beneath the main installation directory, you might want to copy the entire example directory from the J2ME Wireless Toolkit into the Development directory of Forte for Java. This will make the J2ME Wireless Toolkit example MIDlets available from within Forte for Java.

When you first run Forte for Java, it will prompt you to use the J2ME Wireless Compiler with the current project (Figure 3.9). You should click Yes to accept this compiler because you will need it to compile and build MIDlets in Forte for Java.

Note

If you aren't prompted to use the J2ME Wireless Compiler when you first run Forte for Java, it probably wasn't installed correctly and doesn't recognize the J2ME Wireless Toolkit. It is very important that the J2ME Wireless Toolkit be installed *after* you install Forte for Java. Otherwise, the J2ME add-on module won't be properly integrated with Forte for Java and you won't be able to compile or test MIDlets from within Forte for Java. If you've already installed the toolkits in the wrong order, you will need to uninstall them and then reinstall them in the correct order—first Forte for Java, and then the J2ME Wireless Toolkit.

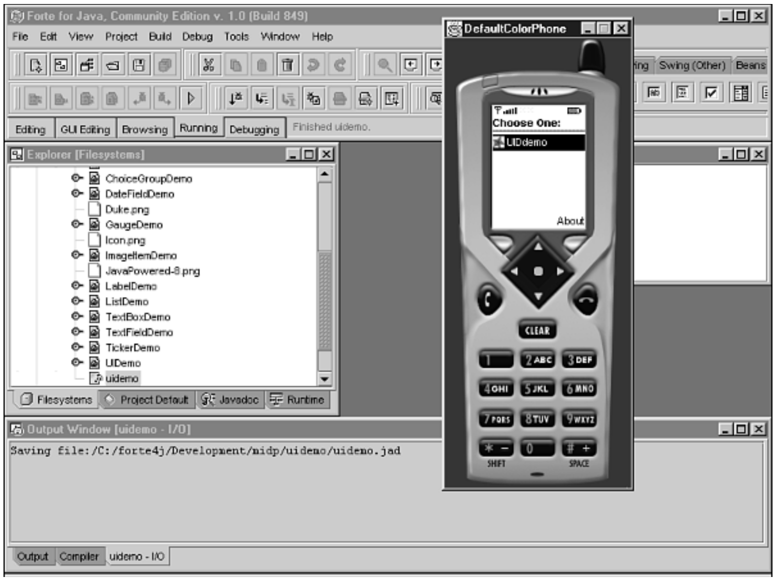
After accepting the J2ME Wireless Compiler for the current project, Forte for Java continues to load and a list of directories will appear in the Forte Explorer window. Click to open the `midp` folder and then the `uidemo` folder. At the bottom of the file list you will see a file named `uidemo`. This is the JAD file for the project, and is used to execute the MIDlet suite. Right-click the file in the list and select Execute from the pop-up menu. Forte for Java will then compile and pre-verify the project if necessary, and then launch the UIDemo MIDlets in the J2ME emulator, as shown in Figure 3.10.


Forte for Java, Community Edition v. 1.0 (Build 849)
File Edit View Project Build Debug Tools Window Help
Editing GUI Editing Browsing Running Debugging
Explorer [Filesystems]
Filesystems
C:\forte4j\Development
example
Tip of the Day
Did you know ... ?
Question
Yes No Never
Next Tip Close
Filesystems Project Default
Properties Window [No Properties]
<No Properties>

J2ME Wireless Module detected that the current project does not use J2ME Wireless compiler. Do you want to set it?

FIGURE 3.9

Upon first running Forte for Java, you are prompted to use the J2ME Wireless Compiler.


Forte for Java, Community Edition v. 1.0 (Build 849)
File Edit View Project Build Debug Tools Window Help
Editing GUI Editing Browsing Running Debugging
Finished uidemo.
DefaultColorPhone
Choose One:
UIDdemo
About
Explorer [Filesystems]
ChoiceGroupDemo
DateFieldDemo
Duke.png
GaugeDemo
Icon.png
ImageItemDemo
JavaPowered-8.png
LabelDemo
ListDemo
TextBoxDemo
TextFieldDemo
TickerDemo
UIDemo
uidemo
Filesystems Project Default Javadoc Runtime
Output Window [uidemo - I/O]
Output Compiler uidemo - I/O

```
Saving file:/C:/forte4j/Development/midp/uidemo/uidemo.jad
```

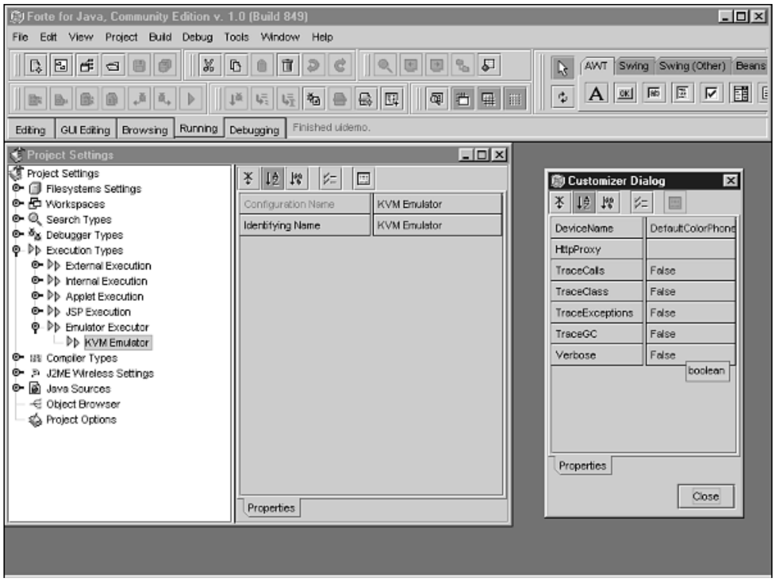
FIGURE 3.10

The UIDemo MIDlet suite is executed in the J2ME emulator from within Forte for Java.

Similar to KToolbar, Forte for Java enables you to change the J2ME emulator properties through a visual interface. To access the emulator properties from within Forte for Java, you must first select Settings from the main Project menu. You will then be presented with the Project Settings dialog box, which contains a list of expandable project settings. Click next to Execution Types to view the different types of execution supported for the project. Then click Emulator Executor to reveal the KVM Emulator entry. Finally, right-click KVM Emulator and select Customize from the drop-down list to display a Customizer Dialog that contains the emulator properties (Figure 3.11).

FIGURE 3.11

The Customizer Dialog in Forte for Java provides access to J2ME emulator properties.



Configuration Name	KVM Emulator
Identifying Name	KVM Emulator

DeviceName	DefaultColorPhone
HttpProxy	
TraceCalls	False
TraceClass	False
TraceExceptions	False
TraceGC	False
Verbose	False

These emulator properties should now be familiar to you because you've seen them a few times throughout the lesson. If you need a refresher, refer to Table 3.2.

Summary

The J2ME emulator is a critical part of J2ME development because it enables you to test MIDlets in your development environment without the need of a physical device. This lesson explored the details of the J2ME emulator and what is has to offer in terms of MIDlet testing. You learned about the standard J2ME emulator that ships with the J2ME Wireless Toolkit, as well as another comparable emulator made available by Motorola. This lesson guided you through the specifics of running the J2ME emulator directly from a command line, as well as from within a visual development environment.

Now that you are comfortable with the use of the J2ME emulator to test MIDlets, you're ready to begin learning how to build MIDlets on your own. The next lesson tackles the basics of MIDlet construction and guides you through the development of your first MIDlet.

Q&A

Q Is it important to always test a MIDlet on several device profiles?

A It depends. The type of device targeted by a MIDlet can vary greatly depending on the MIDlet itself, which means that some MIDlets might be worth testing on a range of devices. On the other hand, some MIDlets might require a large color screen or a full keyboard that is only available on a very specific device. So, to answer the question very vaguely, you must take a close look at how you expect your MIDlet to be used and then assess the device characteristics required of it. After doing that, it is certainly valuable to test the MIDlet on all the device profiles that adhere to your MIDlet's target requirements.

Q Is there an advantage to running the J2ME emulator from the command line versus from within a visual development environment, or vice versa?

A Not exactly. All the features and properties of the J2ME emulator are readily available from both the command line and from within visual development environments. However, where you might find an advantage is the simple convenience of being able to launch the emulator from within a visual development environment without having to issue lengthy commands or doctoring script files.

Q How are the visual development environments able to utilize the command-line J2ME emulator?

A Because the emulator is a command-line application, its output is sent to standard output, which is typically the command window. It is possible for another application to intervene and fill the role of standard output, which is what visual development environments do to receive emulator output. The visual application launches the emulator as a separate process and then routes the output of the emulator to a special window.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What are two parameters of a MIDlet's execution that cannot be accounted for in the J2ME emulator?
2. What J2ME device profile maps the letter keys on the keyboard to device keys?
3. What is the purpose of the J2ME emulator's `kvem.home` system property?
4. What are the pieces of MIDlet information that can be traced by turning on certain emulator properties?

Exercises

1. Try out the J2ME emulator for yourself from the command-line by copying the `run.bat` script file and changing it to use a different device profile.
2. Test out the UIDemo MIDlet example using the KToolBar visual development environment. Think of this as a preview for some of the GUI components you'll be using on Day 8.

DAY 4

Building Your First MIDlet

As anyone who has ever tinkered with Java knows, an applet is a special Java program designed for use within a Web page. Applets must adhere to a certain API that facilitates the merger with Web pages and the execution from within a Web browser. MIDlets are the J2ME equivalent of applets, except MIDlets don't require a Web browser or Web page in order to execute. Instead, a MIDlet only requires a J2ME runtime environment to be available on the mobile devices in which it is run. Similar to applets, MIDlets are relatively easy to develop thanks to the extensive API framework made available in the J2ME APIs.

This lesson tackles the details of developing MIDlets for use on wireless mobile devices. Although the MIDlet you create in this lesson is admittedly somewhat basic, the emphasis is on becoming acquainted with MIDlet construction and learning how to use the appropriate tools to compile, pre-verify, package, and test a MIDlet. The major topics covered throughout this lesson are

- Comparing MIDlets with applets and applications
- Exploring the inner workings of a MIDlet
- Building a complete MIDlet from scratch
- Assessing MIDlet construction within a visual development environment

What Is a MIDlet?

Sun Microsystems has used the "let" suffix to name several different types of programs that fall within the Java technology umbrella. Applets, servlets, spotlets, and now MIDlets are some of these. Sun might have even tried to use the term sniglets had comedian Rich Hall not already worn it out on Saturday Night Live in the 1980's. MIDlets are programs that are developed for use in a mobile computing environment using the J2ME APIs. More specifically, the CLDC and MIDP APIs must be used to develop MIDlets. A MIDlet cannot use classes or interfaces that don't appear in the CLDC and MIDP APIs. This lesson introduces you to some of the important classes found in these APIs, but you get a more detailed analysis in tomorrow's lesson, Day 5, "Working within the CLDC and MIDP APIs."

Unlike an applet, a MIDlet isn't designed to run within a Web browser. This might lead you to think that a MIDlet is more akin to a standalone Java application with a `main()` method. However, unlike applications, MIDlets don't have `main()` methods and are not executed using the Java interpreter. In this regard, MIDlets are closer to applets because a special runtime environment is required for MIDlets to run. This environment primarily consists of an application manager that provides a means of selecting and launching MIDlets on a mobile device. Also similar to applets, the application manager for a MIDlet is responsible for establishing a frame window for the MIDlet. You do not have to explicitly create a frame window for a MIDlet as you must do with graphical standalone applications.

Unlike applets and applications, which are somewhat flexible in terms of how they are packaged and distributed, MIDlets must be packaged in a very specific way for distribution. First, the bytecode classes associated with a MIDlet must go through a pre-verification process that serves as a security optimization to take some of the load off the virtual machine when it loads a MIDlet. The resulting pre-verified MIDlet classes must then be packaged into a Java archive (JAR file) along with a manifest file that describes the MIDlet. This JAR file is then paired with a special application descriptor file (JAD file), which finally readies the MIDlet for distribution.

If the packaging/distribution process for a MIDlet sounds complicated, don't worry because it really isn't too difficult after you step through it with an example. Later in this lesson you work through the development, compilation, pre-verification, packaging, and testing of your first MIDlet.

Inside a MIDlet

You already know how a MIDlet differs from traditional Java applets and applications, but that doesn't really say much about how a MIDlet is structured. It probably won't come as too much of a surprise that every MIDlet must derive from a standard class that is part of the MIDP API. This class is located in the `javax.microedition.midlet` package, and is named `MIDlet`. Although the `MIDlet` class defines several methods, three are particularly important to developing your own MIDlets:

- **`startApp()`**—starts a MIDlet
- **`pauseApp()`**—pauses a MIDlet
- **`destroyApp()`**—destroys a MIDlet

Note

In case you're wondering, the classes and interfaces in the CLDC and MIDP APIs are organized similarly to those in the standard J2SE API. More specifically, the CLDC and MIDP APIs consist of a suite of packages that provide organization to the various classes and interfaces used in MIDlet development. You learn all about these packages in tomorrow's lesson.

To better understand how these methods impact a MIDlet, it's important to clarify that a MIDlet has three distinct states that determine how it functions: *Active*, *Paused*, and *Destroyed*. These states correspond directly with the three methods I just mentioned. In other words, each method places a MIDlet in a given state, which in turn impacts how the MIDlet functions. These methods are usually called directly by the runtime environment's application manager, but in some cases you can call them yourself, particularly the `destroyApp()` method. These methods are collectively referred to as *lifecycle methods* because they control the lifecycle of a MIDlet. They are ultimately what allow the application manager to manage multiple MIDlets and provide each of them shared access to device resources.

NEW TERM *lifecycle methods*—Methods that control the lifecycle of a MIDlet by changing its state.

MIDlet Life Cycle

The lifecycle of a MIDlet is spent between the three states you just learned about. In a typical MIDlet, most of the time is spent in the *Active* or *Paused* states, and then when the MIDlet closes it enters the *Destroyed* state. You override the relevant MIDlet lifecycle methods in most MIDlets because it is important to allocate and free resources based on the state of the MIDlet. For example, when a MIDlet starts you will probably need to create objects and/or load data from a file or network connection. When the MIDlet pauses, it will probably make sense to free up some resources and possibly close a network connection so that another MIDlet can have more memory and device resources with which to work. And finally, upon destruction it is important to free up any resources you've allocated as well as saving any pertinent data to a file or network server.

Keep in mind that a MIDlet can enter and exit the *Active* and *Paused* states several times throughout its lifetime. However, after a MIDlet enters the *Destroyed* state it cannot return. In this regard, an individual MIDlet only has one life to live.

MIDlet Commands

In addition to overriding the lifecycle methods, most MIDlets implement the `commandAction()` method, an event response method defined in the `javax.microedition.lcdui.CommandListener` interface. Commands are used to control MIDlets and initiate actions such as saving data or exiting a MIDlet. MIDlet commands are accessible through a soft button or a menu, and must be handled by a MIDlet using the `commandAction()` method. Handling commands through the `commandAction()` method is similar to handling events in a traditional Java applet or application.

Note

Soft buttons are special buttons located near the screen on a MIDP device that are capable of issuing special commands specific to a particular MIDlet. Clicking a soft button initiates the command on the screen just above the button.

The Display, Screens, and Canvases

One other important MIDlet concept worth tackling at this point is the `Display` class, which represents the display manager for a device. The `Display` class is defined in the `javax.microedition.lcdui` package, along with other GUI classes, and is responsible for managing the display and user input for the device. You don't ever create a `Display` object; you typically obtain a reference to the `Display` object in the constructor for a MIDlet, and then use it to establish the user interface in the `startApp()` method. There is exactly one instance of `Display` for each MIDlet that is executing on a device.

Although there is only one instance of the `Display` object for any given MIDlet, multiple `javax.microedition.lcdui.Screen` objects are possible. A *screen* is a generic MIDlet GUI component that serves as a base class for other important components. The significance of screens is that they represent an entire screen of information; only one screen can be displayed at a time. You can think of multiple screens as a stack of cards that you can flip through. Most MIDlets utilize subclasses of the `Screen` class such as `javax.microedition.lcdui.Form`, `javax.microedition.lcdui.TextBox`, or `javax.microedition.lcdui.List` because they provide more specific functionality.

NEW TERM *screen*—A generic MIDlet GUI component that serves as a base class for other important components.

Another class that is somewhat similar to `Screen` is `javax.microedition.lcdui.Canvas`, which represents an abstract drawing surface the size of the device screen. Unlike a screen, a *canvas* is used to perform direct graphics operations such as drawing lines and curves, or displaying images. `Canvas` objects can be used with `Screen` objects to provide a comprehensive GUI for a MIDlet. More specifically, you can obtain information from the user through a screen containing GUI components, and then render the results on a canvas. You can't show a screen and a canvas at the same time, but you can alternate between the two.

NEW TERM *canvas*—A MIDlet GUI component used to perform direct graphics operations such as drawing lines and curves, or displaying images.

MIDlet Development Revisited

MIDlet development is somewhat different from traditional Java applet and application development, and you need to understand what is involved. Keep in mind that you must have the standard Java SDK and a suitable J2ME development toolkit installed to build and test MIDlets. The development steps involved in taking a MIDlet from concept to reality are

1. Develop the source code files.
2. Compile the source code files into bytecode classes.
3. Pre-verify the bytecode classes.
4. Package the bytecode classes into a JAR file with any additional resources and a manifest file.
5. Develop an application descriptor file (JAD file) to accompany the JAR file.
6. Test and debug the MIDlet.

Step 1 is carried out using a simple text editor. If you don't have a programming text editor, you can use a simple editor such as Notepad in Windows. Step 2 involves using the standard Java compiler to compile the source code files for your MIDlet. In step 3 a J2ME toolkit enters the picture because you are required to pre-verify the compiled bytecode classes using a special pre-verification tool. If you've ever developed JavaBeans components, Step 4 shouldn't be too surprising because it is a similar process for MIDlets. In step 4 you use the Java archive (JAR) tool to compress the MIDlet source code files, resources, and manifest file into a JAR file. Step 5 requires you to create a special application descriptor file, which is a text file containing information about your MIDlet. And finally, in step 6 you get to enjoy your hard work and test out the MIDlet using the J2ME emulator. If you don't remember the J2ME emulator, refer to Day 3, "Getting to Know the J2ME Emulator." Otherwise, let's start creating your very first MIDlet!

Note

Other than its support for outputting trace information, the J2ME emulator doesn't include a debugger. To debug a MIDlet, you'll need a more advanced tool such as a visual development environment with an integrated debugger that supports J2ME. Examples of such tools are Sun's Forte for Java and Metrowerks CodeWarrior for Java.

A Painfully Simple MIDlet

Although I'd love to lead you through the creation of a real-time 3D multiplayer game as your first MIDlet, I don't think it would serve as a best first impression of how MIDlets are structured and coded because of the overwhelming complexity. Instead, you learn how to build a very simple MIDlet that displays a line of text onscreen and handles a common command. I'm sure you've probably seen the familiar "Hello world!" application used to introduce a new programming language. Because I'm from the South (Tennessee), I've opted to change things up a bit with a "Howdy pardner!" MIDlet. Not too many people there actually use that phrase, but I'd hate to destroy such a popular stereotype.

In building the Howdy MIDlet, you'll go through the same sequence of steps outlined in the previous section. This process will be very similar for every MIDlet that you develop. Following are the steps involved in creating the Howdy MIDlet:

1. Code the MIDlet
2. Compile the MIDlet
3. Pre-verify the MIDlet

4. Package the MIDlet
5. Test the MIDlet

The next few sections tackle each of these steps in succession, culminating in the completion of your first J2ME MIDlet.

Note

The next several sections assume that you'll be using the J2ME Wireless Toolkit for MIDlet development. If you've opted to use the Motorola SDK for J2ME or some other command-line toolkit, you will need to account for differences in how the pre-verification tool and emulator are invoked.

Coding the MIDlet

If you've spent much time developing Java applets, developing the code for MIDlets will probably come quite naturally to you. Some structural differences are evident between MIDlets and applets, as you've already learned, but overall the coding of MIDlets is fairly similar to the coding of applets. It's mainly a matter of becoming comfortable with the API classes and interfaces involved in MIDlet development. In this section you assemble the code for the Howdy MIDlet a section at a time, with the complete source code listing appearing at the end.

The first code for your MIDlet is the code that imports a couple of important J2ME packages. Although you can certainly avoid importing any packages and reference every J2ME class and interface using its fully qualified name (`javax.microedition.midlet.MIDlet`, for example), this can be quite cumbersome and ultimately makes the code hard to read. So, the first two lines of your MIDlet import the two primary packages associated with MIDlet development:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

The `javax.microedition.midlet` package includes support for the `MIDlet` class itself, while the `javax.microedition.lcdui` package includes support for the GUI classes and interfaces that are used to construct MIDlet GUIs. With those two packages imported, you're ready to declare the `Howdy` MIDlet class, which is derived from `MIDlet`:

```
public class Howdy extends MIDlet implements CommandListener {
```

The fact that `Howdy` extends `MIDlet` is no surprise, but the implementation of the `CommandListener` interface might seem a little strange. Earlier in the lesson I mentioned that most MIDlets establish commands that enable a user to control a MIDlet and do things such as save data and exit the MIDlet. In the case of the Howdy MIDlet, you will create an Exit command to allow the user to exit the MIDlet. It is necessary for the Howdy class to implement the CommandListener interface so that it can respond to command events.

The Exit command is created as a member variable of the Howdy class, along with a Display object and a special form. Following are the member variables for the Howdy class:

```
private Command exitCommand;
private Display display;
private Form screen;
```

The exitCommand variable is an instance of the javax.microedition.lcdui.Command class that represents a command. The display variable holds a reference to the Display object for the MIDlet. Finally, the screen variable is a javax.microedition.lcdui.Form object that represents the main screen of the MIDlet. All three of these member variables are initialized in the constructor for the Howdy class, which follows:

```
public Howdy() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the Exit command
  exitCommand = new Command("Exit", Command.EXIT, 2);

  // Create the main screen form
  screen = new Form("Howdy");

  // Create a string item and add it to the screen
  StringItem strItem = new StringItem("", "Howdy pardner!");
  screen.append(strItem);

  // Set the Exit command
  screen.addCommand(exitCommand);
  screen.setCommandListener(this);
}
```

The first step of most MIDlets is to obtain a reference to the Display object for the MIDlet. This is carried out by calling the static getDisplay() method on the Display class, and passing in the MIDlet class as the only argument. The Display object is important because it is required for a MIDlet to create its user interface.

The Exit command is created in the Howdy constructor by passing three arguments to the Command constructor that specify the name of the command, its type, and its priority. The name of the command is user-defined and appears above one of the soft buttons on a device or on a menu, depending on its priority and how many buttons are available. The command type must be one of several built-in `Command` constants such as `EXIT`, `OK`, or `CANCEL`.

The priority of a command is used to determine the placement of the command for user access. This is necessary because most devices have limited buttons available for application-specific usage. Therefore, only the most important commands are mapped to these soft buttons. Other commands are still available, but only from menus within the application that aren't as easily accessible as clicking a soft button. The priority value of a command decreases with the level of importance, that is, a value of 1 is assigned to a command with the highest priority. Several commands can have the same priority as long as you realize that the application manager will ultimately decide how they are made available to the user. In the Howdy example, the `Exit` command is given a priority of 2, which means that it has a high priority. Of course, priority values are entirely relative, and because no other commands are in this example the value doesn't matter.

The main screen for the MIDlet of the Howdy constructor is created as a `Form` object. Forms are designed to house other GUI components, and in this case the main screen form is used to hold a `javax.microedition.lcdui.StringItem` component. This component is created with no title, and its text set to `"Howdy pardner!"`. As you might guess, the `StringItem` component is used to display a string of text. However, the text displayed in a `StringItem` component isn't editable. Also, I left the title blank so that it wouldn't be displayed next to the other text.

After creating the `StringItem` object, the code adds the component to the screen by calling the `append()` method on the `Screen` object. This makes the string item a part of the screen, which results in the text being displayed whenever the screen is selected in the current display.

The last section of code in the `Howdy` constructor wires the `Exit` command for the MIDlet. First, the command is added to the screen so that it becomes active. It is still necessary to designate a command listener to receive and process command events. This is accomplished by calling the `setCommandListener()` method and passing `this`, which makes the MIDlet class (`Howdy`) the command listener. That's perfect because earlier you made the `Howdy` class implement the `CommandListener` interface.

With the `MIDlet` constructor out of the way, you can move on to the lifecycle methods. The first of these methods is `startApp()`, which is called whenever the MIDlet enters the *Active* state. Following is the code for the `startApp()` method, which sets the current display to the main Howdy screen.

```
public void startApp() throws MIDletStateChangeException {
  // Set the current display to the screen
  display.setCurrent(screen);
}
```

The `setCurrent()` method sets the current display to the main Howdy screen. This is all the code necessary to display the `screen` object.

The remaining lifecycle methods, `pauseApp()` and `destroyApp()`, aren't required to do anything in the Howdy MIDlet, so their code is quite simple:

```
public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}
```

The last method required of the Howdy MIDlet is the `commandAction()` event response method. If you recall from earlier in the lesson, this method receives command events and must respond to them accordingly. In this case, the only command in the Howdy MIDlet is the `Exit` command, so the `commandAction()` method only has to respond to one command:

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
}
```

The two arguments passed into the `commandAction()` method are the command and the screen on which the command was generated. Only the command is of interest in this particular example. The `Command` object is compared with the `exitCommand` member variable to see whether the Exit command is indeed the command being handled. If so, the `destroyApp()` method is called to destroy the MIDlet. The `false` argument to this method indicates that the destruction is not unconditional, which means that the MIDlet can throw an exception and reject the destruction. Of course, because you wrote the code for the `destroyApp()` method, it's pretty clear that this isn't going to happen. The other option is to pass `true` to `destroyApp()`, which forces the MIDlet to be destroyed no matter what. The `notifyDestroyed()` method is called afterward to notify the application manager that the MIDlet has entered the *Destroyed* state.

That wraps up the code for the Howdy MIDlet. You will gain some perspective on the code by seeing it all together in one listing. So here is the complete code found in the MIDlet source code file `Howdy.java`:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Howdy extends MIDlet implements CommandListener {
  private Command exitCommand;
  private Display display;
  private Form screen;

  public Howdy() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit command
    exitCommand = new Command("Exit", Command.EXIT, 2);

    // Create the main screen form
    screen = new Form("Howdy");

    // Create a string item and add it to the screen
    StringItem strItem = new StringItem("", "Howdy pardner!");
    screen.append(strItem);

    // Set the Exit command
    screen.addCommand(exitCommand);
    screen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the screen
    display.setCurrent(screen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
  }
}
```

Now that the code is complete, you're ready to fire up the Java compiler and compile the MIDlet into a bytecode class file.

Compiling the MIDlet

You use the standard Java compiler to compile MIDlet source code files into bytecode executables. However, a few tricks are involved in the compilation. You must consider that the resulting bytecode will eventually go through a pre-verification process. The significance of this fact is that you will end up with two versions of every class file—the unverified version and the verified version. For this reason, it is helpful to create subdirectories for each set of class files. The J2ME Wireless Toolkit encourages the use of the subdirectories `tmpclasses` and `classes` to hold the unverified and verified class files, respectively. So, you should go ahead and create these directories beneath the directory containing the `Howdy.java` source code file.

With the `tmpclasses` and `classes` directories in place, you're ready to move forward with compilation. The command required to compile the Howdy MIDlet is a little longer than what you're probably accustomed to issuing when compiling a standard Java applet or application:

```
javac -g:none -d tmpclasses -bootclasspath c:\j2mewtk\lib\midpapi.zip
-classpath tmpclasses;classes Howdy.java
```

Let's take each of the command options one at a time. The first option, `-g:none`, informs the compiler not to include any debugging information. The second option, `-d tmpclasses`, tells the compiler where to put the compiled bytecode classes. The third option is perhaps the most important because it overrides the compiler's normal bootstrap class file and tells it to use the MIDP API file instead. Notice that this option is dependent on the installation directory of the J2ME Wireless Toolkit being `c:\j2mewtk`. If you've installed the toolkit somewhere else, be sure to reflect that in this command option. The fourth and final option sets the classpath for the compiler so that the `tmpclasses` directory is included. And last but certainly not least, the actual `Howdy.java` source code file is specified to the compiler.

Note

The Java SDK's `bin` directory must be in the environment path in order to run the standard Java compiler (javac). If you've used the Java SDK before to build Java applets or applications, your path is probably set correctly.

Because the previous compile command that must be issued to compile the Howdy MIDlet is fairly long, I encourage you to place the code in a script and execute the script to compile the MIDlet. As an example, I am working in a Windows environment so I placed the command in a batch file named `Compile.bat` that is located in the same directory as `Howdy.java`. When I make a change to the source code and I need to recompile, I simply double-click `Compile.bat` in Windows Explorer. I use this similar approach in each step of the MIDlet construction process whenever a long command is involved.

Keep in mind that you can also use a visual development environment such as KToolBar or Forte for Java to automate the compilation process. In fact, I will show you how easy it is to compile, pre-verify, package, and test the Howdy MIDlet using the KToolBar tool at the end of this lesson.

Pre-Verifying the MIDlet

The J2ME Wireless Toolkit includes a special tool, pre-verify, that is used to pre-verify MIDlet class files so that the runtime environment doesn't have to do so much work. Because you've already created two directories for holding the MIDlet's class files, you will invoke the preverify tool so that it verifies classes in the `tmpclasses` directories and then writes the verified classes in the `classes` directory. Following is the command to carry this out on the Howdy MIDlet classes:

```
c:\j2mewtk\bin\preverify -classpath c:\j2mewtk\lib\midpapi.zip;tmpclasses
-d classes tmpclasses
```

It is expected that you'll be issuing this command from the same directory where the `Howdy.java` source code file is located, which is the parent directory of the `tmpclasses` and `classes` directories. Because of where you are issuing the command, it is necessary to fully qualify the location of the pre-verify command-line application. Notice in the command that the MIDP API class file is added to the class path, as well as the `tmpclasses` directory. The last option to the tool, `-d classes`, specifies where the pre-verified classes are to be written. The `tmpclasses` argument specifies where to find the source classes to be verified. Upon issuing this command, the MIDlet class file is pre-verified and stored in the classes directory.

Similar to the command used to compile the Howdy MIDlet, I found it convenient to place the `preverify` command in a script to ease its execution. In this case, I named the script `Preverify.bat`.

Packaging the MIDlet

Your MIDlet is now pre-verified and ready for packaging. Packaging a MIDlet requires using the standard Java Archive (JAR) tool that is included in the Java SDK. The JAR tool takes a group of files and compresses them into an archive known as a JAR file. In addition to compressing the pre-verified class files for a MIDlet into a JAR file, you must also include in the JAR file any resources associated with the MIDlet, as well as a manifest file that describes the contents of the JAR file.

In the case of the Howdy MIDlet, the only resource is an icon that is displayed next to the MIDlet name on a device. I will explain about the icon in just a moment. For now, let's focus on the manifest file. The manifest file is a special text file that contains a listing of MIDlet properties and their respective values. This information is very important because it identifies the name, icon, and classname of each MIDlet in a JAR file, as well as the specific CLDC and MIDP version targeted by the MIDlets in the JAR file. Notice that I said MIDlets, which alludes to the fact that multiple MIDlets can be stored in a single JAR file. In fact, MIDlets are often distributed as MIDlet suites that consist of several related MIDlets.

The manifest file for a MIDlet suite must be named `Manifest.mf`, and must be placed in the JAR file with the MIDlet classes and resources. Following is the code for the Manifest file associated with the Howdy MIDlet:

```
MIDlet-1: Howdy, /icons/Howdy.png, Howdy
MIDlet-Name: Howdy
MIDlet-Description: Howdy Example MIDlet
MIDlet-Vendor: Michael Morrison
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

The first entry in the manifest file is by far the most important because it specifies the name of the MIDlet, along with its icon file and its executable classname. You might notice that the property is named `MIDlet-1`; if you were to include additional MIDlets in this MIDlet suite, you would reference them as `MIDlet-2`, `MIDlet-3`, and so on. The `MIDlet-Name`, `MIDlet-Description`, `MIDlet-Vendor`, and `MIDlet-Version` properties all refer to the MIDlet suite as a whole. However, in this case the Howdy MIDlet is the only MIDlet in the suite, so it's okay to refer to the entire suite as Howdy. The last two properties identify the versions of the configuration and profile that the MIDlet suite targets, which in this case are CLDC 1.0 and MIDP 1.0. These properties will become increasingly important as additional configurations and profiles are developed.

Earlier I mentioned that you must include any resources in a JAR file along with the MIDlet classes and the manifest file. At the very least a MIDlet will have an icon as a resource. This icon is simply a 12×12 pixel image that is stored in the PNG image format. It can be a color or black-and-white image, but keep in mind that many mobile devices are black and white. It really depends on your target device when it comes to deciding whether you should create a color icon. I created a small picture of a guy waving to use as the Howdy icon, and stored it in a file called `Howdy.png`.

One small caveat in regard to the icon for a MIDlet is that it must be stored in a directory named `icon` within the MIDlet JAR file. Similar to ZIP files, JAR files enable you to store files within directories inside of the archive. To place a file in such a directory inside a JAR file, you must reference it from a subdirectory when you add it to the JAR file. It is somewhat standard convention to place MIDlet resources in a directory named `res` beneath the directory where the source code files are located. Knowing this, a simple solution for the icon is to place it in an icon directory located beneath the `res` directory. Then, when you create the JAR file you simply specify the `res` directory and the icon will be added with the icon directory information included. To summarize, create an icon and store it in a `res\icon` directory beneath the directory containing the source code files for the MIDlet.

With the icon placed in the proper directory, packaging a MIDlet in a JAR file becomes pretty simple. Following is the code that compresses the Howdy MIDlet into a JAR file with its manifest file and icon:

```
jar cmf Manifest.mf Howdy.jar -C classes . -C res .
```

The first option in this command, `cmf`, indicates that the jar tool is to create a new archive with the specified manifest file (`Manifest.mf`), and also that it is to be named with the specified name (`Howdy.jar`). The `–C` option tells the tool to change to the specified directory and include the specified files. In this case, the directory is `classes` and the file is `.`, which means that all files in the classes directory are to be placed in the archive. The same option is used on the `res` directory to include the MIDlet icon. Even though this command isn't too long, I still recommend placing it in a script so that you can repackage the MIDlet with ease. I named my script `Package.bat`.

At this point the MIDlet is successfully packaged into a JAR file, but a final step is required to test or distribute the MIDlet, the creation of an application descriptor file, or JAD file. This file contains information similar to that found in the manifest file. The Java emulator uses the JAD file when you test a MIDlet suite. In fact, the JAD file is the file you specify to the Java emulator when testing a MIDlet suite. Because you will recognize most of the entries in the JAD file, I show you the JAD file for the Howdy MIDlet:

```
MIDlet-1: Howdy, /icons/Howdy.png, Howdy
MIDlet-Name: Howdy
MIDlet-Description: Howdy Example MIDlet
MIDlet-Vendor: Michael Morrison
MIDlet-Version: 1.0
MIDlet-Jar-Size: 1620
MIDlet-Jar-URL: Howdy.jar
```

The only two surprises here are the last two entries, which specify the size of the JAR file (in bytes) for the MIDlet and the name of the JAR file. It is important that you update the size of the JAR file any time you repackage the MIDlet because the size is likely to change each time you rebuild the MIDlet. Otherwise, the JAD file contains information with which you are already quite familiar. That wraps up the packaging of the MIDlet; you are finally ready to take it for a test drive!

Testing the MIDlet

The previous lesson addressed the J2ME emulator and how it is used to run MIDlets within a virtual device environment. Rather than rehash all that you learned in that lesson, I'll jump straight to the command that runs the Howdy MIDlet using the J2ME emulator:

```
java -cp c:\j2mewtk\lib\kvem.jar;c:\j2mewtk\lib\kenv.zip;
c:\j2mewtk\lib\lime.jar;Howdy.jar -Dkvem.home=c:\j2mewtk com.sun.kvem.midp.Main
DefaultGrayPhone -descriptor Howdy.jad
```

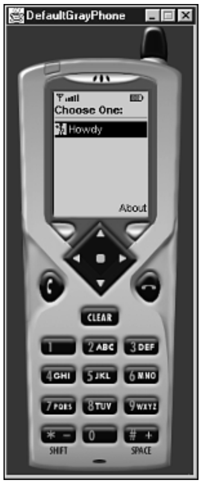
It's quite possible that your command-line environment won't allow a command this long to be issued, so you'll probably be better off placing it in a script. I placed it in a Windows batch file named `Emulate.bat`, which makes it very easy to emulate the MIDlet. Figure 4.1 shows the Howdy MIDlet as it appears in the application manager in the J2ME emulator.

Note

The emulation command for the Howdy MIDlet uses the DefaultGrayPhone device profile. You can easily change the target profile to one of the other standard profiles such as DefaultColorPhone or Pager.

FIGURE 4.1

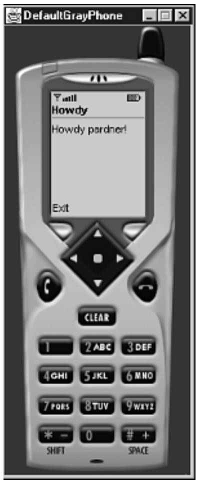
The Howdy MIDlet is made available for running within the application manager of the J2ME emulator.



Because Howdy is the only MIDlet in the MIDlet suite, it is already highlighted and ready to be launched. To run it, click the action button on the device keypad, which appears between the arrow keys; you can also press Enter on your computer keyboard. Figure 4.2 shows the Howdy MIDlet executing in the emulator.

FIGURE 4.2

The Howdy MIDlet displays the simple text message as it is executed in the emulator.



To exit the Howdy MIDlet, click the soft button beneath the word "Exit." This invokes the `Exit` command and results in the MIDlet being destroyed. You can also exit a MIDlet by clicking the red End button, which is used on a real device to end a phone call.

Building and Testing the Howdy MIDlet with KToolBar

As you've seen, building a MIDlet from scratch using command-line tools is pretty straightforward but it does take a little work. If you create a set of scripts to handle all the details of issuing the lengthy commands, the process is greatly simplified. Another way to simplify the MIDlet construction process is to use a visual development environment such as the KToolBar application that comes with the J2ME Wireless Toolkit. KToolBar doesn't include a source code editor, so you still must edit the source code for your MIDlets by hand. However, it makes the compilation, pre-verification, packaging, and testing of a MIDlet completely automatic.

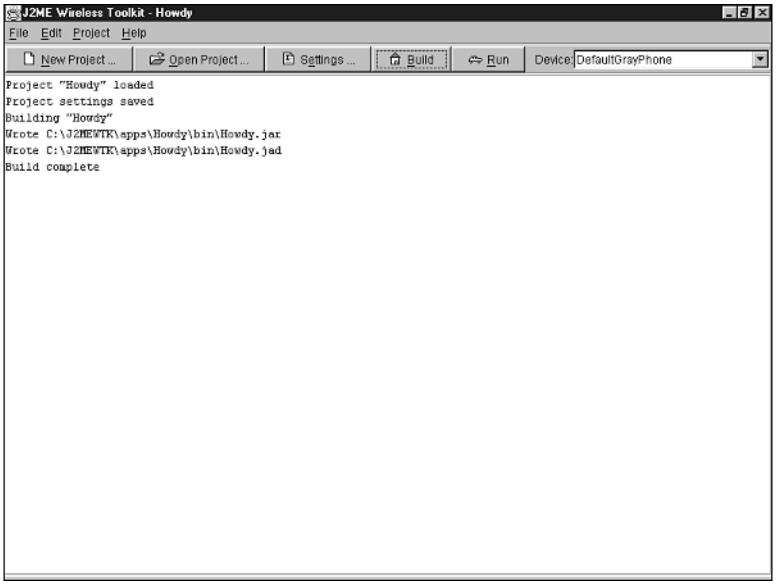
Before using KToolBar with the Howdy MIDlet, it's important to understand that KToolBar requires all MIDlet projects to be located beneath the `apps` directory that is in the J2ME Wireless Toolkit installation directory. You need to copy the directory containing the Howdy MIDlet to the `apps` directory. KToolBar also expects the main directory for a MIDlet to be named the same as the base name of the JAD file. This means that the directory for the Howdy MIDlet must be named Howdy.

Another organizational change required for KToolBar is the placement of the source code files and distribution files for a MIDlet. KToolBar expects the source code files for a MIDlet to be stored in a directory named `src` that is located beneath the main project directory. Also, the manifest file, JAR file, and JAD file for the MIDlet should be placed in a directory named `bin` that is located beneath the main project directory. I know this organizational stuff seems picky, but it is the careful placement of files that enables KToolBar to automate the entire build process.

After the folders and files are properly organized, launch KToolBar and click the Open Project button on the main toolbar. You will be presented with a list of projects that now includes the Howdy project. Select Howdy and click OK to open the project. After the project loads, you can compile, pre-verify, and package it in one swift click of the mouse by clicking the Build button on the toolbar. Figure 4.3 shows the result of building the Howdy project using KToolBar.

FIGURE 4.3

The KToolBar visual development tool makes it possible to build a MIDlet project in one simple step.



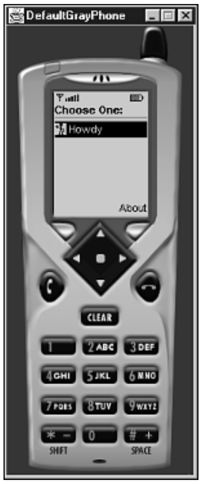
Device: DefaultGrayPhone

```
Project "Howdy" loaded
Project settings saved
Building "Howdy"
Wrote C:\J2MEWTK\apps\Howdy\bin\Howdy.jar
Wrote C:\J2MEWTK\apps\Howdy\bin\Howdy.jad
Build complete
```

After the project is built, you click the Run button to run it using the Java interpreter. Figure 4.4 shows the Howdy MIDlet after being run from the KToolBar tool.

FIGURE 4.4

The Howdy MIDlet can be easily run in the J2ME emulator from the KToolBar tool.



It's pretty obvious how much simpler the development process becomes when you use a visual development environment. On the other hand, with the proper scripts, the command-line approach still isn't too bad. I'll leave it up to you to determine which approach you employ throughout the remainder of the book. Just remember that all of the sample source code is available on the accompanying CD-ROM, including command-line scripts to simplify the construction process.

Summary

In this lesson you finally got to see some Java code for a real MIDlet. Not only that, but you actually built your very first MIDlet from scratch and learned how to use the various tools that are necessary to take a MIDlet from source code files to a distributable JAR file. Before you dove into the coding, however, this lesson made sure to lay the groundwork for what exactly constitutes a MIDlet, and how MIDlets compare to Java applets and applications. You also learned about the lifecycle of a MIDlet and the methods in the `MIDlet` class that manage this lifecycle.

This lesson introduced you to several classes and interfaces that are entirely unique to MIDlet programming. The next lesson gives you some background on these classes and interfaces by exploring the CLDC and MIDP APIs in detail.

Q&A

Q What exactly is the application manager that controls user access to MIDlets?

A The application manager for MIDlets is provided as part of the MIDP runtime environment, and is responsible for providing access to MIDlets that are installed on a given device. The application manager is somewhat of an enabling application for MIDlets, sort of how a Web browser is an enabling application for applets. The application manager takes care of some of the overhead for a MIDlet, such as the window within which a MIDlet displays itself. The application manager also provides a user interface for selecting and launching MIDlets.

Q If you don't provide an `Exit` command for a MIDlet, is it impossible to exit the MIDlet?

A No. The End (red) button on a device is an alternative way to terminate MIDlets. However, I recommend that you always include an Exit command in your MIDlets so users have a very clear and easy way to exit. Additionally, the End button results in the `destroyApp()` method being called with a value of `true`, which means that the MIDlet has no chance to bail out of exiting. The Exit command is handled in the `commandAction()` event response method, which you can design with a little more flexibility in terms of exiting the MIDlet or not.

Q Could the Howdy MIDlet also be implemented so that the "Howdy pardner!" text is drawn graphically on a canvas?

A Yes. In fact, that is how the familiar "Hello world!" example is typically implemented in Java applets. However, in this particular case the code turned out a little simpler by using the `StringItem` GUI component to display the text.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What is the purpose of the pre-verification process that is required of all MIDlet classes?
2. What are the three lifecycle methods associated with the `MIDlet` class?
3. What is the maximum number of `Display` objects that can be associated with a given MIDlet?
4. What is the significance of a MIDlet command's priority?

Exercises

1. Modify the Howdy MIDlet so that it displays your name instead of the text "Howdy pardner!". Rebuild the MIDlet using the command-line tools and then test it in the J2ME emulator.
2. Use the KToolBar visual development tool to rebuild the MIDlet; then run it from within KToolBar but change the device profile to Pager.

DAY 5

Working within the CLDC and MIDP APIs

Java development of any kind revolves around two things: pretzels and beer. Okay, that's not entirely true. The two primary elements involved in Java development are actually the Java programming language and a set of APIs that provide support for application services such as GUI components, networking, and input/output. J2ME development is no different in that it relies on a set of APIs for supporting the various pieces of functionality required for MIDlets that must execute in a wireless mobile environment. The specific APIs required for MIDlet development are outlined in the CLDC and MIDP specifications, which define the configuration and profile for wireless mobile devices. The CLDC and MIDP APIs consist of standard J2SE classes and interfaces, with which you probably already have some familiarity, as well as new classes and interfaces that are entirely unique to MIDlet programming.

This lesson explores the CLDC and MIDP APIs in an attempt to show you the big picture of the classes and interfaces that revolve around MIDlet development. Although this lesson admittedly has a little more of a reference feel than other lessons, you'll still find some practical code sprinkled throughout the lesson. More importantly, this lesson lays out a roadmap for the CLDC and MIDP APIs, which will serve you well as you progress through the remainder of the book and explore how to build MIDlets that use the classes and interfaces found in these APIs. The major topics addressed in this lesson are as follows:

- Understanding why MIDlet development technically involves two APIs
- Becoming acquainted with the classes and interfaces included in the CLDC and MIDP APIs
- Exploring practical uses for classes inherited from the standard J2SE API
- Building a MIDlet that displays information about a device and its runtime environment

MIDlet Programming: A Tale of Two APIs

As you know by now, MIDlets are special programs designed to meet the requirements of the MIDP (Mobile Information Device Profile) specification. The MIDP specification is simply a set of rules that describe the capabilities and limitations of Java with respect to mobile information devices. A significant aspect of these capabilities and limitations is the standard set of classes and interfaces that are available for MIDlet programming. Whereas the MIDP specification provides a detailed description of the API available for MIDlet development, an additional API is provided by the CLDC (Connected Limited Device Configuration). In fact, the MIDP API builds on the CLDC API to provide classes and interfaces that are more specific to mobile information devices. You can think of the CLDC as providing a general Java API for networked devices, whereas the MIDP goes a step further in providing a more detailed API that fills in the specifics left out of the CLDC API for compact wireless devices such as phones and pagers.

Why should you care about any of these specifications and APIs? The CLDC and MIDP specifications are important because they explicitly define what classes and interfaces can be used to build MIDlets. Gone are the days of having a massive API with loads of classes and interfaces to use however you choose. Mobile devices are nimble machines that don't have the luxury of megabytes of memory to pack full of application overhead. Knowing this, Sun had to figure out a way to provide a core set of functionality with a useful feature set but without bloating the runtime requirements of mobile devices. Their answer is the two-tier approach that consists of a configuration layered with a more detailed profile. The CLDC is the configuration used by mobile devices, whereas MIDP is the profile used by mobile information devices.

The CLDC API describes the core classes and interfaces required by a general network device. The MIDP API adds to the CLDC API the classes and interfaces required by a mobile information device. Figure 5.1 shows the relationship between a MIDlet and the respective CLDC and MIDP APIs.

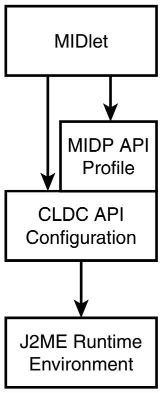


FIGURE 5.1
A MIDlet must make calls to the CLDC and MIDP APIs to carry out most of its functions.

Keep in mind that although the CLDC and MIDP APIs have been carefully thought out to trade off functionality against the memory and resource constraints of mobile devices, they will inevitably come up short in certain situations. There simply is no way to anticipate every possible usage of such an API while also accommodating the limitations of mobile devices. This means that you will sometimes have to work a little harder as a MIDlet developer because you don't have as rich an API to work with as you would if you were doing traditional Java applet or application development.

The CLDC API

Although J2ME programs are structured a little differently from traditional Java applets and applications, they ultimately carry out similar tasks and address many of the same problems as their J2SE and J2EE counterparts. For this reason, the CLDC API is designed as a subset of the standard J2SE API. The majority of the classes in the CLDC API are directly included from the J2SE API. These classes and interfaces are practically identical to those that you are already familiar with from traditional Java programming. This portion of the CLDC API is located in packages with familiar J2SE names such as `java.lang` and `java.util`.

In addition to the classes and interfaces that are borrowed directly from the J2SE API, a few interfaces are unique to the CLDC API. These interfaces deal primarily with networking, which is an area of the J2SE API that is particularly difficult to scale down for the needs of network devices. Granted, the whole idea of network devices is to facilitate network communication, but the networking support in J2SE is so extensive that it comes with a lot of overhead. Additionally, the networking requirements of network devices vary a great deal depending on the specific device. This means that it is really better for the specific networking classes and interfaces for a range of devices to be specified in a profile, as opposed to the configuration. For this reason, the CLDC defines a set of interfaces that facilitate generic networking, and leaves the specifics of implementing these interfaces to the profile (MIDP) API.

So, the CLDC API is logically divided into two parts:

1. A series of packages that serve as a subset of the J2SE API
2. A set of generic networking interfaces

The next few sections explore the classes and interfaces of the CLDC API in more detail.

CLDC and the Standard J2SE API

The bulk of the classes and interfaces in the CLDC API are inherited directly from the J2SE API. Because they come to the CLDC API directly from J2SE, they are organized using the same package conventions as J2SE. Following are the CLDC packages that include classes and interfaces inherited directly from J2SE:

- `java.lang`
- `java.util`
- `java.io`

Note that not all of the classes and interfaces in these packages are included in the CLDC API. Rather, the most important classes and interfaces have made their way to the CLDC API; many others have been left out. J2ME requires that any classes or interfaces inherited directly from J2SE must not be changed in any way, which means that the methods and fields are identical to the versions found in J2SE. This makes it easier to learn how to program in J2ME, and it also makes Java code more portable between J2SE and J2ME.

The next few sections clarify exactly which classes and interfaces are included in each of the three CLDC packages that are inherited from J2SE.

The `java.lang` Package

As you are no doubt aware, the J2SE `java.lang` package includes classes and interfaces that relate to the core Java language. More specifically, this class includes support for Java language features such as data type wrappers, strings, exceptions, and threads, to name a few. In general, you can think of the classes and interfaces in the `java.lang` as having close ties to the Java virtual machine, which makes most of them quite important. This helps explain why so many of them appear in the CLDC API. Following is a list of the CLDC classes and interfaces in the `java.lang` package:

- **`Boolean`**—Wraps the `boolean` primitive data type
- **`Byte`**—Wraps the `byte` primitive data type
- **`Character`**—Wraps the `char` primitive data type
- **`Class`**—Provides runtime information for a class
- **`Integer`**—Wraps the `int` primitive data type
- **`Long`**—Wraps the `long` primitive data type
- **`Math`**—Provides access to several mathematical operations and constants
- **`Object`**—The superclass of all classes in Java
- **`Runnable`**—An interface that provides a means of creating a thread without subclassing the `Thread` class
- **`Runtime`**—Provides access to the Java runtime environment
- **`Short`**—Wraps the `short` primitive data type
- **`String`**—Represents a constant string of text
- **`StringBuffer`**—Represents a string of text with a variable value and length
- **`System`**—Provides access to system resources
- **`Thread`**—Used to create a thread of execution within a program
- **`Throwable`**—Provides support for low-level exception handling

The `java.util` Package

The J2SE `java.util` package includes a variety of classes and interfaces that support different utility functions. For example, classes for working with dates and times, managing data structures, and generating random numbers are a few. The CLDC inherits several of the classes in the J2SE `java.util` package, as the following list shows:

- **`Calendar`**—Provides a means of converting between numeric values and a `Date` object.
- **`Date`**—Represents a system-independent instant in time.

- **`Enumeration`**—An interface that describes a means of iterating through a set of values.
- **`Hashtable`**—A collection that associates keys with values.
- **`Random`**—Provides a simple pseudo-random number generator.
- **`Stack`**—A stack collection consisting of last-in-first-out (LIFO) objects.
- **`TimeZone`**—Represents a time zone offset.
- **`Vector`**—A dynamic array collection.

The `java.io` Package

The J2SE `java.io` package provides classes and interfaces that support the reading and writing of data. Inputting streams of data, outputting streams of data, and working with files are some things that can be done with the classes and interfaces in this package. Not surprisingly, the CLDC doesn't inherit the file support in the J2SE `java.io` package. In fact, the entire task of storing and retrieving persistent data is left to the API for a given profile, which for our purposes is the MIDP API. A variety of I/O classes and interfaces are still included in the CDLC `java.io` package, as the following list shows:

- **`ByteArrayInputStream`**—An input stream that buffers its input in an internal byte array.
- **`ByteArrayOutputStream`**—An output stream that buffers its output in an internal byte array.
- **`DataInput`**—An interface that defines methods to read data from a binary stream into primitive Java data types.
- **`DataInputStream`**—A stream from which data can be read into primitive Java data types.
- **`DataOutput`**—An interface that defines methods to write data from primitive Java data types to a binary stream.
- **`DataOutputStream`**—A stream to which data can be written from primitive Java data types.
- **`InputStream`**—The base class for all input streams.
- **`InputStreamReader`**—A stream from which data can be read as characters of text.
- **`OutputStream`**—The base class for all output streams.
- **`OutputStreamWriter`**—A stream to which data can be written as characters of text.
- **`PrintStream`**—An output stream that facilitates the output of individual primitive data types.
- **`Reader`**—An abstract class for reading character streams.
- **`Writer`**—An abstract class for writing character streams.

The CLDC Generic Connection Framework

It's doubtful that you were surprised by any of the CLDC classes and interfaces inherited from the J2SE API. Most of those classes are common Java classes that are encountered fairly regularly in traditional applet and application programming. Where the CLDC veers away from the J2SE API is in its support for networking. I mentioned earlier the difficulties involved in providing networking support at the configuration level given the variety of different networking approaches employed by network devices. Because of these difficulties, the CLDC pushes most of the specifics of network support onto the profile for a certain type of device. To be able to successfully delegate this responsibility, the CLDC outlines a generic network framework known as the *Generic Connection Framework* (*GCF*).

The purpose of the GCF is to define a general network architecture that supports networked I/O that is extremely flexible, and therefore extensible. Implementing a generic architecture at the CLDC level is done because there is too much overhead involved in inheriting the necessary networking classes and interfaces from the J2SE API. Additionally, it is difficult to predict what networking support a specific device might require. Because you ideally don't want to include anything that a device doesn't need, it's best to let the profile for each major device type address networking specifics. This puts the CLDC in the position of providing a consistent framework that is open enough to support a wide range of networking options.

NEW TERM *Generic Connection Framework(GCF)*—A generic network framework that is extremely flexible and extensible, and is open-ended enough to handle the wide range of network connections and protocols used in conjunction with network devices.

Interestingly enough, the GCF is still designed as a functional subset of the J2SE networking classes, which means that features described in the GCF are available in J2SE. However, the extensible architecture of the GCF necessitates a different hierarchy of classes and interfaces than what is found in J2SE. The GCF consists primarily of a set of connection interfaces, along with a `Connector` class that is used to establish the different connections. Both the `Connector` class and the connection interfaces are located in the `javax.microedition.io` package. Following are the connection interfaces found in this package:

- **`Connection`**—A basic connection that can only be opened and closed.
- **`ContentConnection`**—A stream connection that provides access to Web data.
- **`DatagramConnection`**—A datagram connection suitable for handling packet-oriented communication.

- **`InputConnection`**—An input connection to a communications device.
- **`OutputConnection`**—An output connection to a communications device.
- **`StreamConnection`**—A two-way connection to a communications device.
- **`StreamConnectionNotifier`**—A special notification connection that is used to wait for a stream connection to be established.

These interfaces form the Generic Connection Framework (GCF), and therefore don't directly provide any usable functionality. Instead, it is expected that a profile will take these interfaces and provide the capability of establishing real network connections with them. You learn about the MIDP classes and interfaces that complement these interfaces in the next section of this lesson. You also learn a great deal about how to work with connections on Day 9, "Exploring MIDlet I/O and Networking." For now, the emphasis is on getting you acquainted with the CLDC API and its relationship to both the J2SE API and MIDlet networking.

The MIDP API

A device profile picks up where a configuration leaves off by providing detailed functionality to carry out important tasks on a given type of device. In the case of the Mobile Information Device Profile (MIDP), the type of device is a wireless mobile device such as a mobile phone or pager. It is therefore the job of the MIDP API to take the CLDC API and build on top of it the necessary classes and interfaces that make it possible to build compelling MIDlets.

Similar to the CLDC API, the MIDP API can be divided into two parts:

1. Two classes inherited directly from the J2SE API
2. A series of packages that include classes and interfaces unique to MIDP development

The following sections explore the classes and interfaces of the MIDP API in more detail.

MIDP and the Standard J2SE API

Similar to the CLDC API, the MIDP API borrows from the standard J2SE API. However, the degree to which the MIDP API borrows is significantly less than the CLDC API. More specifically, only two MIDP classes are inherited directly from the J2SE API:

- **`Timer`**—Provides scheduling functionality for creating timed tasks
- **`TimerTask`**—Represents a task that is scheduled using the `Timer` class

As you may know, these classes are part of the standard J2SE `java.util` package. This means that they also appear within the `java.util` package that is part of the CLDC/MIDP API. The two timer-related classes are used to schedule tasks so that they occur after a certain amount of time. For example, you could develop an alarm clock MIDlet that uses the timer classes to implement a snooze button.

MIDP-Specific Classes and Interfaces

Not surprisingly, the bulk of the MIDP API is new classes and interfaces designed specifically for use in MIDlet programming. Although these classes and interfaces can play a similar role as some of the classes and interfaces in the J2SE API, they are entirely unique to the MIDP API and therefore are carefully designed to solve MIDlet-specific problems. This portion of the MIDP API is divided among several packages, all of which are prefixed with the `javax.microedition` name:

- `javax.microedition.midlet`
- `javax.microedition.lcdui`
- `javax.microedition.io`
- `javax.microedition.rms`

The following sections examine the classes and interfaces that are found in each of these packages. If you're feeling overwhelmed with all these API descriptions, just hang in there, because toward the end of the lesson you will learn how to do some practical things with the CLDC and MIDP APIs.

The `javax.microedition.midlet` Package

The `javax.microedition.midlet` package is the central package in the MIDlet API, and contains only one class: the `MIDlet` class. You are already familiar with the `MIDlet` class from the previous lesson when you built your first MIDlet. As you learned in that lesson, the `MIDlet` class provides the basic functional overhead required of a MIDP application (MIDlet) that can execute on a mobile device. You will continue to learn more about the `MIDlet` class as you progress through the book and construct more complex MIDlets.

The `javax.microedition.lcdui` Package

The `javax.microedition.lcdui` package includes classes and interfaces that support GUI components that are specially suited for the small screens found in mobile devices. This package is functionally equivalent to the Abstract Windowing Toolkit (AWT) package in the J2SE API, which includes extensive support for GUI components. Because of the constrained nature of mobile devices, however, the `javax.microedition.lcdui` package is significantly smaller than the J2SE AWT package, `java.awt`.

A fundamental concept of MIDlet development is that of a screen, which is a generic MIDlet GUI component that serves as a base class for other important components. A screen serves as the basis for high-level GUI components within the MIDP API. You build a screen GUI by adding other components to it to form a complete user interface. In addition to the high-level screen approach offered by the MIDP API, you can also access a drawing surface and draw graphics directly on the display. This type of drawing surface is known as a canvas, and offers you much more control over the look of a MIDlet than the screen approach.

Of course, the cost of this added control is added complexity and in many cases more development work. The type of MIDlet you are developing will generally dictate one of the two GUI approaches, simplifying the decision process. In terms of the MIDP API, the classes and interfaces for both GUI approaches are included in the `javax.microedition.lcdui` package. Following are the interfaces defined in this package:

- **`Choice`**—An interface that describes a series of items from which the user can choose.
- **`CommandListener`**—An event listener interface for handling high-level commands.
- **`ItemStateListener`**—An event listener interface for handling item state events.

Note

The "lcdui" part of the package name `javax.microedition.lcdui` stands for LCD User Interface. LCD refers to the Liquid Crystal Displays that are commonly used in mobile information devices, which are significantly smaller than typical computer screens. Therefore, the name "lcdui" refers to user interface features designed specifically for use in small LCD screens.

In addition to these interfaces, also several classes are in the `javax.microedition.lcdui` package:

- **`Alert`**—A screen that displays information to the user, and then disappears.
- **`AlertType`**—Represents different types of alerts used with the `Alert` class.
- **`Canvas`**—A low-level drawing surface that allows you to draw graphics on the screen.
- **`ChoiceGroup`**—Presents a group of selectable items; used in conjunction with the `Choice` interface.
- **`Command`**—Represents a high-level command that can be issued from within a MIDlet.

- **`DateField`**—Presents a date and time so that it can be edited.
- **`Display`**—Represents a mobile device's display screen, and also handles the retrieval of user input.
- **`Displayable`**—An abstract component that is capable of being displayed; other GUI components subclass this class.
- **`Font`**—Represents a font and its associated font metrics.
- **`Form`**—A screen that serves as a container for other GUI components.
- **`Gauge`**—Displays a value as a percentage of a bar graph.
- **`Graphics`**—Encapsulates two-dimensional graphics operations such as drawing lines, ellipses, text, and images.
- **`Image`**—Represents a graphical image.
- **`ImageItem`**—A component that supports the layout and display of an image.
- **`Item`**—A component that represents an item with a label.
- **`List`**—A component consisting of a list of choices that can be selected.
- **`Screen`**—Represents a full screen of high-level GUI information, and serves as the base class for all MIDP GUI components.
- **`StringItem`**—A component that represents an item consisting of a label and an associated string of text.
- **`TextBox`**—A type of screen that supports the display and editing of text.
- **`TextField`**—A component that supports the display and editing of text; unlike `TextBox`, this component can be placed on a form with other components.
- **`Ticker`**—A "ticker tape" component that displays text moving horizontally across the display.

You will learn a great deal about these MIDP GUI interfaces and classes on Day 7, "Building Graphical MIDlets," and Day 8, "Making the Most of MIDlet GUIs," when you tackle MIDlet graphics and GUI construction.

The `javax.microedition.io` Package

You already know that the CLDC lays the groundwork for networking and I/O with the `java.io` package and the Generic Connection Framework (GCF). The MIDP API builds on this support with the addition of the `HttpConnection` interface, which is included in the `javax.microedition.io` package. The `HttpConnection` interface solidifies the GCF with support for an HTTP connection, which is useful for accessing Web data from within a MIDlet. You learn more about this interface and how it is used on Day 9.

The `javax.microedition.rms` Package

Because few mobile devices have hard drives or any tangible file system, you probably won't rely on files to store away persistent MIDlet data. Instead, the MIDP API describes an entirely new approach to store and retrieve persistent MIDlet data: the *Record Management System* (*RMS*). The MIDP RMS provides a simple record-based database API for persistently storing data. The classes and interfaces that comprise the RMS are all located in the `javax.microedition.rms` package. Following are the RMS interfaces found in this package:

- **`RecordComparator`**—An interface used to compare two records.
- **`RecordEnumeration`**—An interface used to iterate through records.
- **`RecordFilter`**—An interface used to filter records according to certain criteria.
- **`RecordListener`**—An event listener interface used to handle record change events.

In addition to these interfaces, a few classes fit into the RMS architecture. Most importantly, the `RecordStore` class represents a *record store*, which is essentially a simplified database for MIDlet data storage. The following classes are included in the `javax.microedition.rms` package:

- **`InvalidRecordIDException`**—Thrown whenever an operation fails because the record ID is invalid
- **`RecordStore`**—Represents a record store
- **`RecordStoreException`**—Thrown whenever an operation fails because of a general exception
- **`RecordStoreFullException`**—Thrown whenever an operation fails because the record store is full
- **`RecordStoreNotFoundException`**—Thrown whenever an operation fails because the record store cannot be found
- **`RecordStoreNotOpenException`**—Thrown whenever an operation is attempted on a closed record store

NEW TERM *record store*—A simplified database for MIDlet data storage.

The MIDP RMS is a very convenient way to store and retrieve data for a MIDlet without having to hassle with the complexities of files. It's also significant because it's your only option for storing persistent data locally on a device. The RMS provides a simple database system for storing and retrieving individual chunks of data (records), as well as for filtering and sorting the data. However, the RMS is not a relational database system, which means that you can't create multiple tables of data that are linked to each other.

The primary goal of the RMS is to provide a means of storing MIDlet data persistently. You learn all about the inner workings of the RMS and how to use it in practical MIDlets on Day 13, “Using the MIDP Record Management System (RMS).”

Putting the APIs to Use

You may be convinced that this lesson has turned into some kind of Dewey Decimal look at the J2ME APIs. I admit that it has been a little “referency” thus far, but things will change very quickly now. You probably noticed that several of the MIDP-specific API topics are tackled in greater detail later in the book. For this reason, I won’t spend any more time on them in this lesson. I will spend a little time revisiting some of the classes and interfaces that are borrowed from the J2SE API. You may already have some knowledge of these classes and interfaces, in which case you should be able to breeze through the next few sections. However, if you don’t then I encourage you to study the code snippets and gain some level of comfort with them because you’ll be utilizing much of this knowledge throughout the remainder of the book.

Following are the API topics that are addressed in the next few sections:

- Data type wrappers
- The runtime environment
- Date and time

After quickly covering some of the ways in which these topics are addressed in Java code in the next few sections, you put them to practical use in a MIDlet that displays information about the system such as the current time, the available memory, and display properties.

Data Type Wrappers

The data type wrappers are included in the `java.lang` package, and provide object versions of the following primitive Java data types: `boolean`, `byte`, `char`, `int`, `long`, and `short`. The most popular usage of the wrapper classes is to aid in converting back and forth between primitive types and strings. Following is an example of how to convert a literal integer value to a string representation:

```
int weight = 180;
String strWeight = Integer.toString(weight);
```

New Term *data type wrapper*—A class that provides an object-oriented layer around a primitive data type.

This code enables you to easily convert numeric values for display purposes. This technique comes in handy a little later in the lesson when you build an example MIDlet that displays information about the runtime environment for a device.

You can also convert from a string back to a primitive data type using the wrapper classes. Following is code to carry this out for a string containing a `long` number:

```
String strMile = "5280l";
long mile = Long.parseLong(strMile);
```

The Runtime Environment

The `java.lang.Runtime` class provides access to the runtime environment. More specifically, the `Runtime` class enables you to determine how much total memory is available in the runtime environment, as well as how much of that memory is free for use. To retrieve this information you must first get a `Runtime` object instance, which is accomplished through the static `getRuntime()` method. Following is an example of how this is done:

```
Runtime runtime = Runtime.getRuntime();
```

After you have a `Runtime` object, you can call the `totalMemory()` and `freeMemory()` methods to get the total and free memory for the runtime environment, respectively. Both of the methods return the memory values as a `long` data type. If you want to display the memory values you must first convert them to strings using the `Long` data wrapper class. Following is an example of obtaining the total and free memory, and then converting them to strings:

```
String totalMem = Long.toString(runtime.totalMemory());
String freeMem = Long.toString(runtime.freeMemory());
```

Date and Time

The `Calendar` class that is located in the `java.util` package makes it possible to retrieve the current date and time. To obtain an instance of the `Calendar` object that is initialized to the current date and time, call the static `getInstance()` method, like this:

```
Calendar calendar = Calendar.getInstance();
```

After you have an instance of a `Calendar` object, you can call the `get()` method to retrieve individual date and time properties. These properties are defined as public constants in the `Calendar` class. Following are some of the more commonly used properties associated with the `get()` method:

- **`YEAR`**—The year
- **`MONTH`**—The month of the year

- **`DAY_OF_MONTH`**—The day of the month
- **`DAY_OF_WEEK`**—The day of the week
- **`HOUR`**—The hour of the day
- **`MINUTE`**—The minute within the hour
- **`SECOND`**—The second within the minute
- **`MILLISECOND`**—The millisecond within the second

These properties enable you to retrieve individual portions of a date or time. As an example, the following code shows how to retrieve the current hour of the day as an integer between 0 and 23:

```
int hour = calendar.get(Calendar.HOUR);
```

By retrieving several different pieces of the time, converting them to strings, and concatenating them appropriately, you can assemble a string containing the time in HH:MM:SS form. Following is an example of code that accomplishes this task:

```
String time = Integer.toString(calendar.get(Calendar.HOUR)) + ":" +
              Integer.toString(calendar.get(Calendar.MINUTE)) + ":" +
              Integer.toString(calendar.get(Calendar.SECOND));
```

The SysInfo MIDlet

Thus far you've explored a few portions of the standard J2SE APIs that are included in the CLDC/MIDP APIs. Because this book is obviously about J2ME programming, you might have been a little concerned that I was digressing unnecessarily into too much J2SE material. Now it's time to pull together these seemingly unrelated topics into another example MIDlet. The example MIDlet is called SysInfo, and it builds on the Howdy MIDlet from the previous lesson by displaying several pieces of information about the system and runtime environment. More specifically, the SysInfo MIDlet displays the following information pertaining to a mobile device:

- The current time in HH:MM:SS form
- The amount of total available memory
- The amount of free memory
- Whether the display supports color
- The number of colors (or shades of gray) supported by the display

This can seem like a lot of information, but the code for the MIDlet demonstrates how easy it is to obtain this information and convert it into a form suitable for display. Speaking of the SysInfo MIDlet code, here it is:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class SysInfo extends MIDlet implements CommandListener {
  private Command exitCommand;
  private Display display;
  private Form screen;

  public SysInfo() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit command
    exitCommand = new Command("Exit", Command.EXIT, 2);

    // Create the main screen form
    screen = new Form("SysInfo");

    // Obtain the current time
    Calendar calendar = Calendar.getInstance();
    String time = Integer.toString(calendar.get(Calendar.HOUR)) + ":" +
                  Integer.toString(calendar.get(Calendar.MINUTE)) + ":" +
                  Integer.toString(calendar.get(Calendar.SECOND));

    // Obtain the total and free memory, and convert them to strings
    Runtime runtime = Runtime.getRuntime();
    String totalMem = Long.toString(runtime.totalMemory());
    String freeMem = Long.toString(runtime.freeMemory());

    // Obtain the display properties
    String isColor = display.isColor() ? "Yes" : "No";
    String numColors = Integer.toString(display.numColors());

    // Create string items and add them to the screen
    screen.append(new StringItem("", "Time: " + time));
    screen.append(new StringItem("", "Total mem: " + totalMem));
    screen.append(new StringItem("", "Free mem: " + freeMem));
    screen.append(new StringItem("", "Color: " + isColor));
    screen.append(new StringItem("", "# of colors: " + numColors));

    // Set the Exit command
    screen.addCommand(exitCommand);
    screen.setCommandListener(this);
  }
  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the screen
    display.setCurrent(screen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
  }
}
```

This code might look somewhat familiar as the Howdy MIDlet code with some additions to the constructor. The first interesting addition to the `SysInfo` constructor is the code that builds a string containing the current time in HH:MM:SS form. This code uses the `Calendar` class and the `Integer` data wrapper class to obtain the appropriate time fields and assemble them into a string:

```
Calendar calendar = Calendar.getInstance();
String time = Integer.toString(calendar.get(Calendar.HOUR)) + ":" +
              Integer.toString(calendar.get(Calendar.MINUTE)) + ":" +
              Integer.toString(calendar.get(Calendar.SECOND));
```

In addition to the current time, the SysInfo MIDlet also displays the amount of total and free memory in the runtime environment. The code to obtain this information and convert it to strings makes use of the `Runtime` class and the `Long` data wrapper class:

```
Runtime runtime = Runtime.getRuntime();
String totalMem = Long.toString(runtime.totalMemory());
String freeMem = Long.toString(runtime.freeMemory());
```

The final pieces of information displayed by the SysInfo MIDlet are whether the device display is a color display and how many colors the display can show. The code that assesses these display capabilities uses several methods defined in the `Display` class. More specifically, the `isColor()` method returns a `boolean` value that indicates the display's support for color. Also, the `numColors()` method returns the number of colors that can be shown by the display. Following is the code that retrieves this information in the SysInfo MIDlet:

```
String isColor = display.isColor() ? "Yes" : "No";
String numColors = Integer.toString(display.numColors());
```

Note

In the case of a grayscale display, the number of colors reported by the `numColors()` method indicates the number of shades of gray.

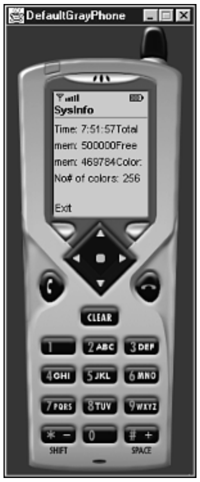
The last bit of new code in the SysInfo MIDlet is the code that actually displays the information that has been gathered. If you recall, in the Howdy MIDlet a single `StringItem` component was used to display a short text greeting. In the SysInfo MIDlet, multiple `StringItem` components are used to display each piece of information. Following is the code in the `SysInfo` constructor that creates these components:

```
screen.append(new StringItem("", "Time: " + time));
screen.append(new StringItem("", "Total mem: " + totalMem));
screen.append(new StringItem("", "Free mem: " + freeMem));
screen.append(new StringItem("", "Color: " + isColor));
screen.append(new StringItem("", "# of colors: " + numColors));
```

Nothing fancy here, just a few descriptive strings put together and used to initialize the string items. Each of these string items is displayed vertically down the display of the device. You can verify this by testing the MIDlet in the emulator, as shown in Figure 5.2.

FIGURE 5.2

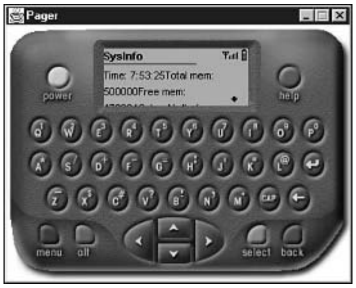
The SysInfo MIDlet displays various pieces of information about the mobile device and its runtime environment.



```
SysInfo
Time: 7:51:57Total
mem: 500000Free
mem: 469784Color:
No# of colors: 256

Exit
```

To check that the string items are arranged when the display changes size, it's worth taking a look at the MIDlet as it appears within the Pager device in the emulator. Figure 5.3 shows the SysInfo MIDlet being run in the emulator using the Pager profile.

The string items are still arranged vertically on the Pager display, but because of the decreased height of the display it is now necessary to scroll down to see all of the information. Although this is probably acceptable, it is important to consider situations such as this when deciding how to present information to the user. If you intend to target several types of devices, it is very important to take care that information is easily accessible to all users.



```
SysInfo
Time: 7:53:25Total mem:
500000Free mem:
```

FIGURE 5.3
Although the size of the display changes when running the SysInfo MIDlet using the Pager profile, the information is still oriented vertically.

Summary

This lesson covered a lot of territory, some of which was admittedly a little like a mini reference for the CLDC and MIDP APIs. The purpose for outlining all the classes and interfaces found in these APIs is to take the mystery out of MIDlet programming so that you can know for certain what you have to work with in terms of classes and interfaces. After spending some time getting acquainted with the classes and interfaces that comprise the CLDC and MIDP APIs, you took a look at some example code snippets that showed a few practical tasks that could be accomplished with classes inherited from the J2SE API. These snippets were then put together near the end of the lesson to form a complete MIDlet that displays information about a device and its runtime environment.

Day 6, “Creating Custom Device Profiles,” shifts gears a bit by addressing custom device profiles and how they are created. It is the conclusion of Part I of the book, and reveals how flexible MIDlet development can be by guiding you through the creation of an entirely custom device profile. Granted, I don’t expect you to be building custom MIDP devices in your garage, but the ability to tweak device profiles provides you with an excellent strategy for stress-testing MIDlets.

Q&A

Q Is there a distinction between the CLDC and MIDP APIs when coding MIDlets?

A No. The distinction between the CLDC and MIDP APIs is purely architectural. In terms of developing MIDlet code, the classes and interfaces defined in the CLDC and MIDP APIs are combined in a single API that consists of several packages. Even so, the architectural distinction is important because it helps to explain the limitations of MIDlets. The packages in the CLDC/MIDP APIs include familiar J2SE package names as well as entirely new J2ME package names.

Q Will Sun ever release different configurations or profiles with a different mix of classes and interfaces?

A Yes. The CLDC and MIDP APIs covered in this lesson are applicable to a certain type of network device. The whole purpose of breaking down the J2ME architecture into configurations and profiles is to facilitate the creation of APIs that address the unique needs of different devices. This book targets the MIDP device profile, which is specifically aimed at wireless mobile devices such as phones and pagers.

Q The amount of total memory reported by the SysInfo MIDlet seemed somewhat large. Is it possible to change the amount of memory available to the emulator?

A No. One weakness of the current release of the J2ME emulator is that it doesn't allow you to change the amount of memory available to MIDlets.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. Which classes in the standard J2SE `java.awt` package are inherited in the CLDC/MIDP APIs?
2. In addition to several connection interfaces, what class is defined in the Generic Connection Framework (GCF) to help establish connections?
3. In what package is the all-important `MIDlet` class located?
4. What is a record store, and what MIDP package contains support for record stores?

Exercises

1. Test the SysInfo MIDlet using the color phone profile, and notice how the display information changes to reflect the color display.
2. Add code to the SysInfo MIDlet so that it also displays the current date in the form MM/DD/YY. Hint: This involves using the `Calendar` class and several of the property constants associated with its `get()` method.

DAY 6

Creating Custom Device Profiles

One interesting aspect of J2ME is its design for use in such a wide range of devices. The MIDP specification narrows the focus of J2ME to a specific type of wireless mobile device, but much room still remains for device manufacturers to create devices with a variety of different features. For this reason, it is important to be able to emulate different types of devices. You've already seen how the J2ME emulator enables you to test a MIDlet on several different devices, but you haven't learned much about how to go beyond emulating the built-in devices that come with the emulator.

This lesson introduces device profiles, which are software descriptions of physical hardware devices. Device profiles play an extremely important part in the J2ME emulator because they describe the parameters of a given device for emulation purposes. When you select a certain device for emulation, you are really selecting its device profile for use with the emulator. This lesson explains exactly what goes into a device profile and how to create custom profiles from scratch. The major topics you work through in this lesson are as follows:

- Understanding the significance of device profiles
- Using the Configuration Editor to create custom device profiles
- Altering a device profile to work with the J2ME emulator
- Testing a device profile within the J2ME emulator

Understanding Device Profiles

Because MIDlets can be run on a variety of different physical devices, it is advantageous to test them on as many of those devices as possible. You already know that emulation is an important part of the MIDlet development process. Consequently, running a MIDlet on various devices within the emulation environment is an important step toward creating robust MIDlets. This obviously requires that the J2ME emulator support the emulation of different devices, which fortunately it does. In fact, the J2ME emulators included in both the J2ME Wireless Toolkit and the Motorola SDK for J2ME enable you to emulate custom devices with properties you can define yourself.

The key to supporting the emulation of multiple devices is the *device profile*, a software description of a physical MIDP device. The J2ME emulator uses a device profile to determine device characteristics such as the image of the device, the location and size of the screen, and more technical details such as the amount of available memory and the size of network data packets. Device profiles enable device manufacturers to introduce new devices and immediately allow developers to begin targeting them without having access to the actual physical devices. Although this is beneficial to device manufacturers and developers alike, one of the most interesting facets of device profiles is that they enable you to tinker with device settings to see how they affect your MIDlets.

NEW TERM *device profile*—A software description of a physical MIDP device (also known as a *device configuration*).

Note

It's worth pointing out that the term "device profile" as used in this lesson is very different from the term "profile" as used when describing a class of devices under the CLDC. A profile is a specific classification of devices that falls under the CLDC; MIDP is an example of such a profile. A device profile is a description of a specific type of device as it is represented within the J2ME emulator. In other words, a device profile has meaning only when referring to the emulation of devices with the J2ME emulator.

From a MIDlet development standpoint, device profiles enable you to experiment with different types of device properties, even those that might not yet exist in a physical device. For example, throughout this chapter I will show you how to create a hypothetical Nintendo Game Boy Color device profile. Although there currently is no physical Game Boy device that supports J2ME, it is an interesting sample device to work with. Besides, game companies are starting to jump on the Java bandwagon, so a Java-powered Game Boy might not be all that far fetched. Even though the Game Boy device doesn't support J2ME, its screen size and button layout are sufficiently similar to a typical MIDP device.

Note

Although the standard J2ME emulator that ships with the J2ME Wireless Toolkit makes use of device profiles to describe different devices, it doesn't include any special tool for creating or editing the profiles. You can create custom profiles that can be used with the J2ME Wireless Toolkit, but they must be coded by hand, which is somewhat tedious. For this reason, this lesson focuses on creating and editing device profiles for use with the Motorola SDK for J2ME. Unfortunately, device profiles currently aren't portable across the different J2ME toolkits. You'll have to stick with the Motorola SDK for J2ME to take advantage of the device profile created in this lesson.

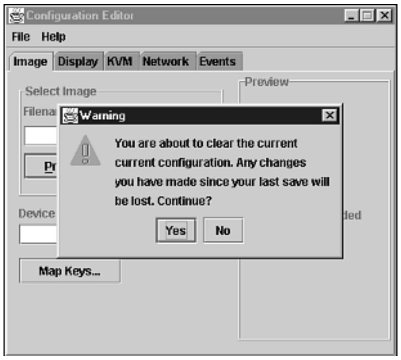
The Motorola SDK for J2ME includes a special tool known as the *Configuration Editor*, which enables you to create and edit device profiles through an intuitive graphical user interface. After specifying all the properties of a device in the Configuration Editor, a special device properties file is generated with a `.props` file extension. This file serves as the device profile and is used by the J2ME emulator to determine the properties of the device for emulation purposes. The `.props` file generated by the Configuration Editor consists of a list of property values that describe each aspect of a device in detail.

Using the Configuration Editor

The Configuration Editor that ships with the Motorola SDK for J2ME enables you to create and edit device profiles for use with the emulator included in the same toolkit. Any device profiles that you create with the Configuration Editor can be used only with the Motorola J2ME emulator and therefore aren't compatible with the standard J2ME emulator that ships with the J2ME Wireless Toolkit. The remainder of this lesson assumes that you're working within the Motorola SDK for J2ME.

The Configuration Editor is a standalone Java application that must be executed using a Java interpreter. The Motorola SDK for J2ME includes a special script that alleviates the need to launch the Configuration Editor from a command line. In the Windows release of the SDK this script is called `runConfig.bat`. Running this script launches the Configuration Editor.

To become acquainted with the Configuration Editor, you will work through the creation of a custom device profile from scratch. The Configuration Editor makes this process very simple, so you'll actually do very little work. To get started, first launch the Configuration Editor and start a new configuration. Run the `runConfig.bat` script and, when the Configuration Editor appears, select New from the File menu. The window shown in Figure 6.1 will appear. Click the Yes button to clear the profile and start a new one.



You are about to clear the current current configuration. Any changes you have made since your last save will be lost. Continue?

FIGURE 6.1
It's important to create a new device profile (configuration) when you first start the Configuration Editor.

Note

For some reason, the Motorola SDK for J2ME interchanges the terms "device profile" and "device configuration;" keep in mind that they mean the same thing.

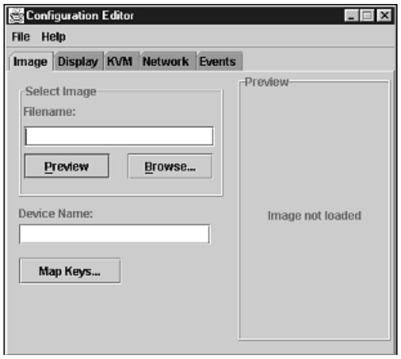
Setting the Display Properties and Key Mappings

With a new profile created and ready to roll, you can start entering information about the device. In this example you're creating a profile for a Nintendo Game Boy Color, which is a hypothetical handheld video game device. You can think of this Game Boy device profile as a bit of wishful thinking on my part; it sure would be a lot of fun to develop MIDlets to run on a Game Boy!

In the Configuration Editor, you'll notice that its user interface is divided into multiple tabs, with each tab targeting a certain aspect of the device. The first is the Image tab, shown in Figure 6.2.

FIGURE 6.2

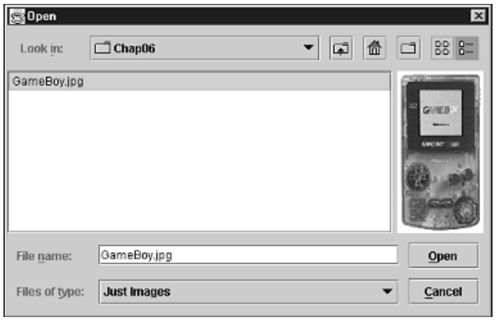
The Image tab of the Configuration Editor enables you to set the image that represents the device.



In the Image tab of the Configuration Editor you can set the image that represents the physical device in the emulator. This image must be stored in the JPEG image format. To select the image, click the Browse button and locate the image on your hard drive using the Open window (see Figure 6.3).

FIGURE 6.3

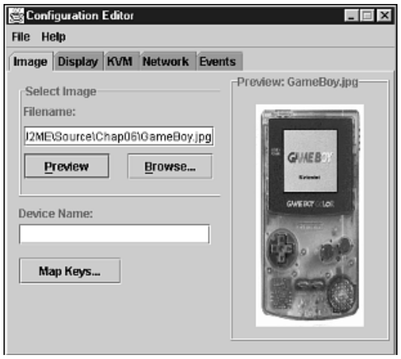
The Open window enables you to browse and select an image for use as the device image.



Note

The JPEG image for a device must be exactly the same size as the physical device. This is important because you will set the screen size of the device in pixels, and the emulator will automatically display the screen on top of the device image. If the device image isn't the correct size, the screen will appear too large or too small. One easy way to see whether the device image is close to the correct size is to select the screen part of the image and see whether it matches the display resolution of the physical device.

After you've found the image on your hard drive, click the Open button to accept the image. In the Image tab of the Configuration Editor, you can click the Preview button to see a preview of the device image (see Figure 6.4).



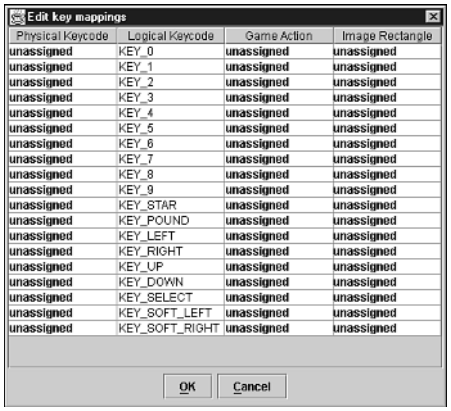
Filename: J2ME\Source\Chap06\GameBoy.jpg

Device Name:

FIGURE 6.4
By clicking the Preview button, you can see a preview of the device image.

You might notice that the Image tab also includes a text box for entering the device name. This is a descriptive name for the device, which in this case should be set to `Nintendo Game Boy`.

The last piece of information required in the Image tab are the mappings for the device keys. Keep in mind that the emulator not only displays the output of a MIDlet on the screen of the device image, it also mimics the functionality of keys that appear in the image. For this to work, you must identify the keys within the image as well as their functionality. You must point out which key serves as the Up Arrow key, which key serves as the Select key, and so on. To establish these key mappings, click the Map Keys button in the Image tab, which displays the Edit Key Mappings window (see Figure 6.5).



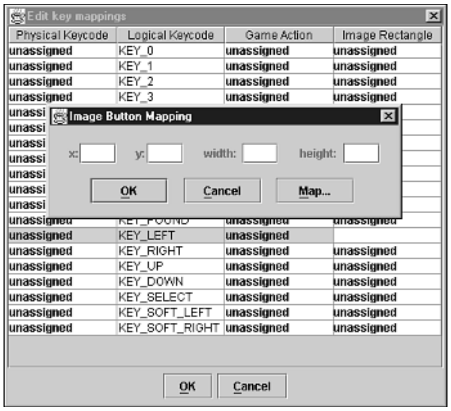
Physical Keycode	Logical Keycode	Game Action	Image Rectangle
unassigned	KEY_0	unassigned	unassigned
unassigned	KEY_1	unassigned	unassigned
unassigned	KEY_2	unassigned	unassigned
unassigned	KEY_3	unassigned	unassigned
unassigned	KEY_4	unassigned	unassigned
unassigned	KEY_5	unassigned	unassigned
unassigned	KEY_6	unassigned	unassigned
unassigned	KEY_7	unassigned	unassigned
unassigned	KEY_8	unassigned	unassigned
unassigned	KEY_9	unassigned	unassigned
unassigned	KEY_STAR	unassigned	unassigned
unassigned	KEY_POUND	unassigned	unassigned
unassigned	KEY_LEFT	unassigned	unassigned
unassigned	KEY_RIGHT	unassigned	unassigned
unassigned	KEY_UP	unassigned	unassigned
unassigned	KEY_DOWN	unassigned	unassigned
unassigned	KEY_SELECT	unassigned	unassigned
unassigned	KEY_SOFT_LEFT	unassigned	unassigned
unassigned	KEY_SOFT_RIGHT	unassigned	unassigned

FIGURE 6.5
The Edit Key Mappings window enables you to establish key mappings for the device configuration.

The Edit Key Mappings window consists of a list of key codes that identify functional keys capable of being supported on a device. Not all devices will support all of the key codes listed, and some devices might support additional key codes not listed. It's pretty obvious that our Game Boy device won't support the numeric keys found on most phones.

NEW TERM *key code*—A functional key that is capable of being supported on a device.

Each line in the Edit Key Mappings window is considered a key mapping because it associates a logical key code with a physical device key. The two columns of primary interest in this list are the second column (the logical key code) and the last column (the rectangle of the physical key within the device image). To establish a mapping for a given key code, you must select the rectangular dimensions of the physical key on the device image. Double-click the Image Rectangle entry for the key you'd like to map. Figure 6.6 shows the result of double-clicking the image rectangle for the `KEY_LEFT` key code.



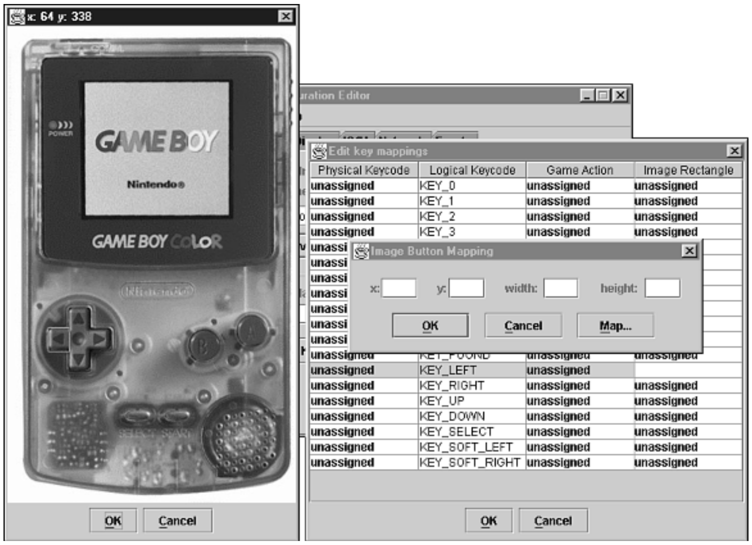
Physical Keycode	Logical Keycode	Game Action	Image Rectangle
unassigned	KEY_0	unassigned	unassigned
unassigned	KEY_1	unassigned	unassigned
unassigned	KEY_2	unassigned	unassigned
unassigned	KEY_3	unassigned	unassigned
unassi			
unassi			
unassi			
unassi			
unassi			
unassi			
unassi			
unassigned	KEY_POUND	unassigned	unassigned
unassigned	KEY_LEFT	unassigned	
unassigned	KEY_RIGHT	unassigned	unassigned
unassigned	KEY_UP	unassigned	unassigned
unassigned	KEY_DOWN	unassigned	unassigned
unassigned	KEY_SELECT	unassigned	unassigned
unassigned	KEY_SOFT_LEFT	unassigned	unassigned
unassigned	KEY_SOFT_RIGHT	unassigned	unassigned

FIGURE 6.6

The Image Button Mapping window enables you to specify the region within the device image for a button mapping.

When you double-click the image rectangle for a key code, the Image Button Mapping window opens. This window enables you to specify the rectangular region within the device image that represents the given key. This rectangle is identified by an x,y coordinate for its upper-left corner, along with a width and height. The x,y coordinate is measured in pixels from the upper-left corner of the device image. Although it's certainly possible to figure out this information using an image-editing tool, the simplest way to do it is to click the Map button and select the rectangle visually, as shown in Figure 6.7.

To select a button visually, click and drag the mouse to draw a rectangle around the portion of the device image containing the button. In the figure I've drawn a rectangle around the left side of the Game Boy game controller, which serves as the key mapping for the `KEY_LEFT` key code. To accept the rectangle, click the OK button, which results in the dimensions of the rectangle being entered into the Image Button Mapping window.



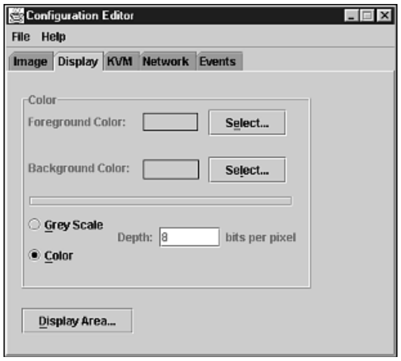
Physical Keycode	Logical Keycode	Game Action	Image Rectangle
unassigned	KEY_0	unassigned	unassigned
unassigned	KEY_1	unassigned	unassigned
unassigned	KEY_2	unassigned	unassigned
unassigned	KEY_3	unassigned	unassigned
unassi			
unassi			
unassi			
unassi			
unassi			
unassi			
unassi			
unassigned	KEY_POUND	unassigned	unassigned
unassigned	KEY_LEFT	unassigned	
unassigned	KEY_RIGHT	unassigned	unassigned
unassigned	KEY_UP	unassigned	unassigned
unassigned	KEY_DOWN	unassigned	unassigned
unassigned	KEY_SELECT	unassigned	unassigned
unassigned	KEY_SOFT_LEFT	unassigned	unassigned
unassigned	KEY_SOFT_RIGHT	unassigned	unassigned

FIGURE 6.7

By clicking the Map button, you can draw the rectangle around a button to be mapped on the device image.

Clicking the OK button in this window accepts the button mapping. The button mapping steps should then be repeated for all the key codes that apply to the device. In the Game Boy device, the following key codes apply: `KEY_LEFT`, `KEY_RIGHT`, `KEY_UP`, `KEY_DOWN`, `KEY_SELECT`, `KEY_SOFT_LEFT`, and `KEY_SOFT_RIGHT`.

The button mapping wraps up the information required for the Image tab of the Configuration Editor. The next tab of interest is the Display tab, which is shown in Figure 6.8.



Depth: 8 bits per pixel

○ Grey Scale

● Color

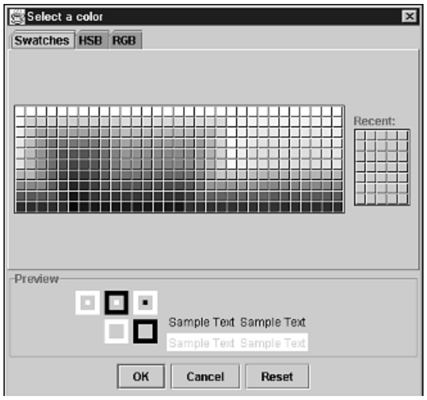
FIGURE 6.8

The Display tab of the Configuration Editor enables you to set the color settings of the display, along with the display area of the device.

The first settings on the Display tab are the foreground and background colors of the display. In addition to approximating the look of the physical device screen, these colors determine the colors of graphics and text in MIDlets. When you consider that mobile device screens can range from greenish-gray LCD screens to bright white full-color screens, the foreground and background colors make a little more sense. The Game Boy screen is an LCD screen that has the greenish-gray appearance, so you need to set the background color accordingly. Likewise, the main foreground color of the Game Boy screen is just a shade off of black. To select these colors, click the Select button next to each color. This opens the Select a Color window, which contains several tabs that enable different approaches to selecting a color (see Figure 6.9).

FIGURE 6.9

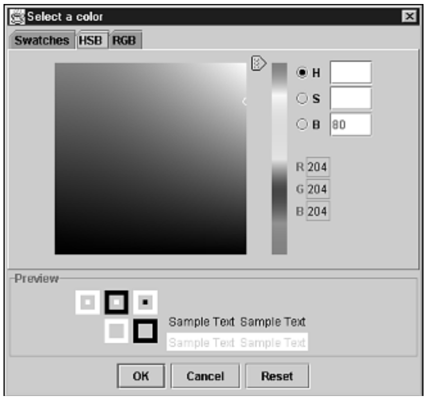
The Select a Color window enables you to select a color visually for the foreground or background display color.



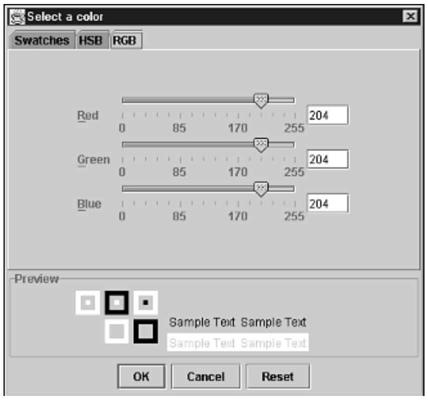
The Select a Color window contains several tabs to select a color using a variety of different techniques. The default Swatches tab enables you to select from a set of predetermined color swatches. To exercise more control over the selection of a color, click the HSB tab, which enables you to enter the hue, saturation, and brightness of a color, as shown in Figure 6.10.

FIGURE 6.10

The HSB tab of the Select a Color window enables you to select a color based on its hue, saturation, and brightness.



By altering these three color components, you can create virtually any color. Another approach for describing colors is RGB, which stands for red, green, and blue. Figure 6.11 shows the RGB tab of the Select a Color window.



Red 204
Green 204
Blue 204

FIGURE 6.11
The RGB tab of the Select a Color window enables you to select a color based on its red, green, and blue components.

Because I played with a lot of Play-Doh as a kid, I tend to think in terms of mixing colors, so the RGB approach to selecting colors makes the most sense to me. Following are some suitable RGB color settings for the foreground and background colors of the Game Boy display:

- **Foreground**—Red = 32, Green = 32, Blue = 32
- **Background**—Red = 188, Green = 198, Blue = 188

After selecting these colors, you're ready to set a few more attributes of the display. More specifically, you need to change the Greyscale setting to Color and set the number of bits per pixel to 5. This doesn't exactly match the color capabilities of a Game Boy Color device, but it's close.

Note

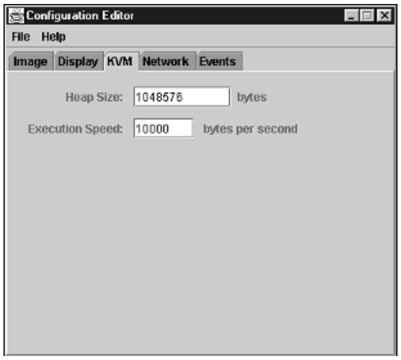
The number of bits per pixel for a device display indicates how many different colors or shades of gray can be shown on the display at any given time. To give a few examples, a black-and-white display is considered a 1-bit display, and a 16-color display is considered a 4-bit display. In the case of the Game Boy Color, the 5-bit setting indicates that the device can display 32 colors at once, which is pretty close to the display characteristics of the real device.

The last step in establishing the display properties of a device is to set the dimensions of the display with respect to the device image. You must specify exactly what portion of the device image is usable as its display. This is important because the emulator uses this rectangular area as the basis for displaying the output of MIDlets executed on the device. To set the display area, click the Display Area button. You will see a window that enables you to enter the x,y coordinates and width and height for the display area, much as you did earlier when mapping buttons. You can then click the Map button to locate the rectangle for the display area visually. Then you will probably need to change the width and height to the exact display setting, which for the Game Boy is 160×140. You can do this by entering these numbers into the Width and Height text boxes. This means that the main benefit of visually locating the display area is in establishing its upper-left corner.

With the display settings for your custom device profile finished, you're ready to move to a couple more issues. Clicking the KVM tab in the Configuration Editor leads you to the next order of business.

Establishing the Available Memory and Execution Speed

The KVM tab of the Configuration Editor is used to set the available memory and execution speed of the device profile. Figure 6.12 shows this tab.



Heap Size: 1048576 bytes

Execution Speed: 10000 bytes per second

FIGURE 6.12

The KVM tab of the Configuration Editor enables you to set the heap size and execution speed of the device configuration.

The available memory that you set in the KVM tab is actually the heap size, which is the amount of working memory available to MIDlets running within the virtual machine of the device. This memory setting is specified in bytes and is useful in testing MIDlets that have a tendency to hog memory. The available memory of a Game Boy is 32KB, or 32,000 bytes, which is what you should enter in the Heap Size text box.

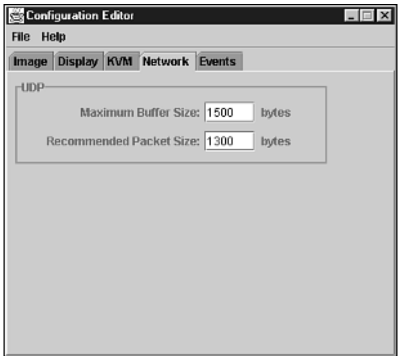
The execution speed isn't supported in the current release of the Motorola J2ME emulator, but it will eventually be used to simulate the speed of a device. The execution speed is measured in the number of bytes of Java bytecode executed by the virtual machine in one second. The idea behind setting this device property is to simulate different processor speeds and make sure that a MIDlet performs as expected on each. For now, because the emulator ignores this setting, leave the default value.

Determining the Networking Capabilities

The Network tab of the Configuration Editor enables you to set the networking capabilities of the device. Figure 6.13 shows this tab, which includes a couple of settings that affect the network characteristics of the device.

FIGURE 6.13

The Network tab of the Configuration Editor enables you to set the maximum buffer size and recommended packet size for UDP network communications.



Maximum Buffer Size: 1500 bytes
Recommended Packet Size: 1300 bytes

The Network tab consists of two properties that determine the network characteristics of the device profile. These properties apply specifically to User Datagram Protocol (UDP) networking, a network protocol that involves sending data in fixed-size packets called *datagrams*. UDP is a popular protocol for performing network communication where the delivery isn't time critical. The maximum buffer size specified in the Network tab determines the maximum size of the memory buffer that stores network data as it is being read. This buffer size is specified in bytes and is important for MIDlets that involve UDP networking. The default value of the maximum buffer size is sufficient for the Game Boy device profile.

NEW TERM *datagram*—A packet of information sent across a network using the UDP protocol.

The other property in the Network tab is the recommended packet size of UDP datagram packets, in bytes. This recommended packet size is more of a request than a hard specification because the underlying networking software can override it if necessary. The default value of the recommended packet size is adequate for the Game Boy device profile.

Note

It's worth noting that not all devices support UDP networking, so these properties might not apply to all devices. HTTP networking is the only type of J2ME networking that is guaranteed to be supported in all devices.

That wraps up the Network tab, leaving you with one more to go. The Events tab of the Configuration Editor enables you to add event support to a device for mouse and key repeat events.

Supporting Events

The Events tab of the Configuration Editor includes a list of events that can be enabled for the device. Figure 6.14 shows this tab, which reveals the events that can be enabled for the device.

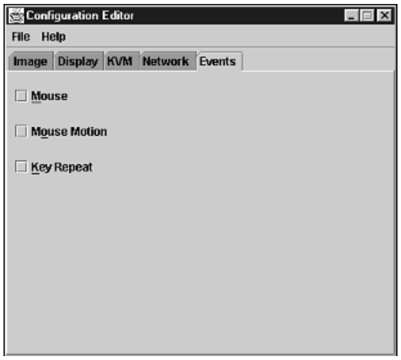


FIGURE 6.14
The Events tab of the Configuration Editor enables event generation for mouse, mouse motion, and key repeat events.

The events in the Events tab correspond to input features that might not be supported in the device. More specifically, you should only enable these events if the device supports a mouse or keys with a repeat function. A repeating key is one that dispatches multiple key-press events when held down. Most MIDP devices don't support any of these events, including the Game Boy device. You can leave them unchecked in the Game Boy device profile.

Finishing Up the Device Profile

With all aspects of the Configuration Editor taken into consideration, your newly created device profile is now ready to roll. However, you do need to save the file so that you can use it within the J2ME emulator. To save the properties file for the device profile, select Save from the File menu, and then enter the name of the properties file with a `.props` file extension. I named the Game Boy properties file `GameBoy.props`.

Now you have a properties file that is almost ready to be used with the Motorola J2ME emulator. A couple of quirks require you to make a few small changes directly to the properties file before testing the device profile. Exit the Configuration Editor and take a look inside the device profile you just created.

Tweaking a Device Profile

As you now know, a device profile created with the Configuration Editor is really just a text file with `.props` file extension. This properties file contains a list of device properties and their respective values as they were set using the Configuration Editor. The current release of the Motorola SDK for J2ME is quirky and doesn't enable a newly created properties file to work seamlessly with the J2ME emulator. For some reason a few properties that the emulator is counting on are missing. You'll need to add them here to smooth things over. To get started, take a look at the complete listing of the `GameBoy.props` properties file:

```
#Device Properties: 2001.02.12
#Mon Feb 12 16:55:28 CST 2001
mousemotionsupport=false
devicename=Nintendo Game Boy
mousesupport=false
bytecodespersecond=10000
keysoftleft=190,316,29,32
iscolordisplay=true
keyrepeatsupport=false
keyleft=40,312,22,25
foregroundcolor=-14671840
keyup=64,293,20,22
keypound=-1,-1,-1,-1
keydown=63,335,22,23
imagename=C\:\\Books\\TYJ2ME\\Source\\Chap06\\GameBoy.jpg
colordepth=5
keysoftright=240,301,30,31
keyright=84,312,23,25
keyselect=118,397,34,15
key9=-1,-1,-1,-1
key8=-1,-1,-1,-1
key7=-1,-1,-1,-1
displayarea=80,58,160,140
key6=-1,-1,-1,-1
key5=-1,-1,-1,-1
udpminpacketsize=1500
```

```
udpmaxpacketsize=1300
key4=-1,-1,-1,-1
key3=-1,-1,-1,-1
key2=-1,-1,-1,-1
heapsize=1048576
key1=-1,-1,-1,-1
backgroundcolor=-4405572
key0=-1,-1,-1,-1
keystar=-1,-1,-1,-1
```

The problem with the properties file that you created with the Configuration Editor is that a few of the property names don't match those required by the Motorola J2ME emulator. Although you can probably assume that this inconsistency will be fixed eventually, for now you must manually make a few changes to the properties file for things to work properly. The first change has to do with the network settings for the device, which involve the `udpminpacketsize` and `udpmaxpacketsize` properties in the `GameBoy.props` file. These two properties must be removed and replaced by the following two properties:

```
maxudpbuffersize=1500
recommendedudppacketsize=1300
```

Note

In the `GameBoy.props` properties file the `imagename` property is set to an exact path on the hard drive, which happens to be unique for my J2ME development setup. This path will be different when you create a properties file on your own computer.

This seems like a strange thing to change, but it has to do with an inconsistency between the Configuration Editor and the J2ME emulator. Again, hopefully it will be resolved in a future release of the Motorola SDK for J2ME. Another change in the properties file is the addition of a new network property called `tcpnumsockets`, which determines the number of TCP/IP sockets that can be used simultaneously. A suitable default value for this property is `5`, which results in the following code:

```
tcpnumsockets=5
```

The last change in the `GameBoy.props` file has to do with the display area, which was set visually in the Configuration Editor. The dimensions you set were stored in a property named `displayarea`, which doesn't work properly with the J2ME emulator. This property must be renamed `screenSize`, as shown in the following code:

```
screensize=80,58,160,140
```

That finishes up the necessary changes required for the device profile to work with the Motorola J2ME Emulator. You're probably anxious to try out your new Game Boy in the emulator, so let's get to it.

Testing a Custom Device Profile

Integrating a custom device profile into the Motorola J2ME emulator is very straightforward, but it does require you to do a few things to get the emulator ready for the device profile. The following three pieces of information are necessary to use a custom device profile with the emulator:

- The properties file for the device profile
- The device image file
- An execution script for the device profile

You already have the first two pieces of information squared away, but the third might seem a little strange. The execution script to which I refer is necessary to keep you from entering a lengthy command at the command line every time you want to run the emulator with the device profile. You must create a short script that invokes the emulator and identifies the `GameBoy.props` file as the device profile to be used. Fortunately, the Motorola SDK for J2ME includes a template for such a script in the `scripts` directory. The following is a modified version of the template script that is suitable for the Game Boy device profile:

```
@if NOT "%1"=="" set MYCLASS=%1
@if     "%1"=="" set MYCLASS=com.mot.j2me.midlets.paddleball.PaddleBall

set CLASSPATH=

set J2ME_RESOURCE_DIR=c:\MotoSDK\lib\resources

cd ..\bin
java -Djava.library.path=../lib -classpath ./Emulator.jar;
➥./ConfigTool.jar com.mot.tools.j2me.emulator.Emulator -classpath
➥../demo/midlets;../lib -deviceFile resources/GameBoy.props
➥javax.microedition.midlet.AppManager %MYCLASS% -JSA 1 1
cd ..\scripts
```

Note

If you are a Windows user, understand that a script is the same thing as a MS-DOS batch file.

This script is named `runGameBoy.bat` and invokes the emulator through the Java interpreter. The significant part of this script is that it specifies the `GameBoy.props` file by using the `-deviceFile` option. The remainder of the script is similar to the other device scripts included in the Motorola SDK for J2ME. You can specify a MIDlet as the argument to the script. It will run the PaddleBall MIDlet if none is specified.

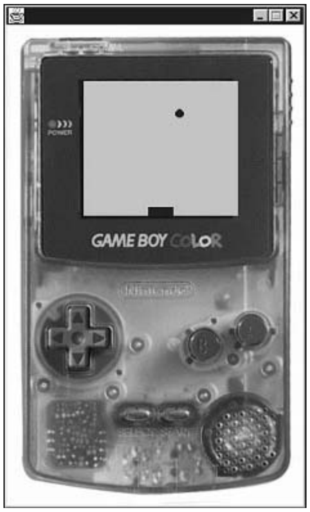
Before you can run this script, you must place the Game Boy device profile files in the appropriate locations within the Motorola SDK for J2ME file structure. More specifically, you need to place the three device profile files in the following directories beneath the Motorola SDK for J2ME installation directory:

- `GameBoy.props   bin\resources`
- `GameBoy.jpg     bin\resources\images`
- `runGameBoy.bat  scripts directory`

After placing these files in the appropriate directories, you're ready to test out the Game Boy device profile. Run the `runGameBoy.bat` script to do so. Figure 6.15 shows the new device as it appears in the J2ME emulator.

FIGURE 6.15

After a custom device configuration has been created, you can test it using the J2ME emulator.



You can test the device by using the B button (Right Soft button) to start the Paddle Ball game, as well as the left and right arrows of the game pad to control the paddle. Keep in mind that you can also test other MIDlets with the Game Boy device by specifying the MIDlet as the command-line argument to the `runGameBoy.bat` script.

Summary

This lesson showed you how to extend the capabilities of the J2ME emulator by incorporating new devices. You began the lesson by learning exactly what a device profile is and why it is important to MIDlet development.

You were introduced to the Configuration Editor, a tool included with the Motorola SDK for J2ME that enables you to create and edit device profiles visually. Although the Configuration Editor is a very slick tool, you learned that it doesn't currently work seamlessly with the Motorola emulator, which means you had to make a few manual changes to the device profile for it to work properly with the emulator. Nevertheless, your work was rewarded when you got to see a MIDlet running on the custom device profile in the emulator.

The next lesson changes direction considerably by introducing you to MIDlet graphics. In addition to learning about the graphical capabilities of MIDlets, you also develop a few MIDlets that demonstrate many of these capabilities.

Q&A

Q Unless I plan to manufacture my own hardware device that supports J2ME, why do I need to know how to create custom device profiles?

A Custom device profiles provide the ultimate in flexibility when it comes to stress testing a MIDlet. Although it is certainly important to test MIDlets on device profiles that correspond to real devices, by tweaking an existing device profile or creating a new one entirely you can carefully test certain aspects of MIDlets and see how they respond in extreme environments.

Q Can I use a custom device profile that I've created with the Configuration Editor in the J2ME Wireless Toolkit?

A No. The current release of the J2ME Wireless Toolkit doesn't directly support custom device profiles. You must stick with the Motorola SDK for J2ME if you're going to use custom device profiles.

Q What device profiles are included in the Motorola SDK for J2ME?

A The Motorola SDK for J2ME includes standard device profiles for several of their phones, including iDEN i1000, iDEN Platform, and StarTac, as well as a generic phone. The StarTac device profile is included purely as an example because the StarTac phone currently has no J2ME support.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What is a device profile?
2. What type of file is generated by the Configuration Editor tool?
3. Why is it necessary to define the dimensions of a device's display area as part of the device profile?
4. Which three files are necessary to use a custom device profile within the Motorola SDK for J2ME?

Exercises

1. Try altering the display area, foreground color, and background color of the Game Boy device profile, and then test it in the Motorola J2ME emulator.
2. Modify the `runGameBoy.bat` script so that it will run the SysInfo MIDlet from the previous lesson. You'll need to add the directory containing the class file to the class path for the emulator to find the SysInfo MIDlet class. The Motorola emulator is different from the J2ME Wireless emulator in that it doesn't use a JAD file.

PART II

Digging Deeper into MIDlet Development

7 Building Graphical MIDlets

8 Making the Most of MIDlet GUIs

9 Exploring MIDlet I/O and Networking

10 Being Your Own Wireless Meteorologist

11 Wireless Navigation

12 Optimizing MIDlet Code

DAY 7

Building Graphical MIDlets

MIDP devices are all relatively constrained in terms of their display capabilities, but in some situations it is beneficial to draw custom graphics within a MIDlet. Games are a great example of MIDlets that require low-level graphics. Another example is a charting MIDlet that keeps up with real-time sales figures from a data server. Such a MIDlet would probably require a graphical display of data. Graphics are an important part of MIDlet programming, even if they don't enter the picture with every MIDlet. The standard MIDP GUI components will be sufficient for most MIDlets, but for those that require a higher degree of graphical customization, you'll want to take advantage of the low-level graphics capabilities of the MIDP API.

These capabilities are referred to as low-level because, although drawing graphics in J2ME is very similar to drawing them in traditional Java programs, it does require more work than relying on the standard MIDP GUI components.

This lesson introduces you to a few classes that make it possible to draw graphics in MIDlets. You also work through a couple of sample MIDlets that demonstrate how to draw lines, curves, text, and images. The following are the major topics covered in this lesson:

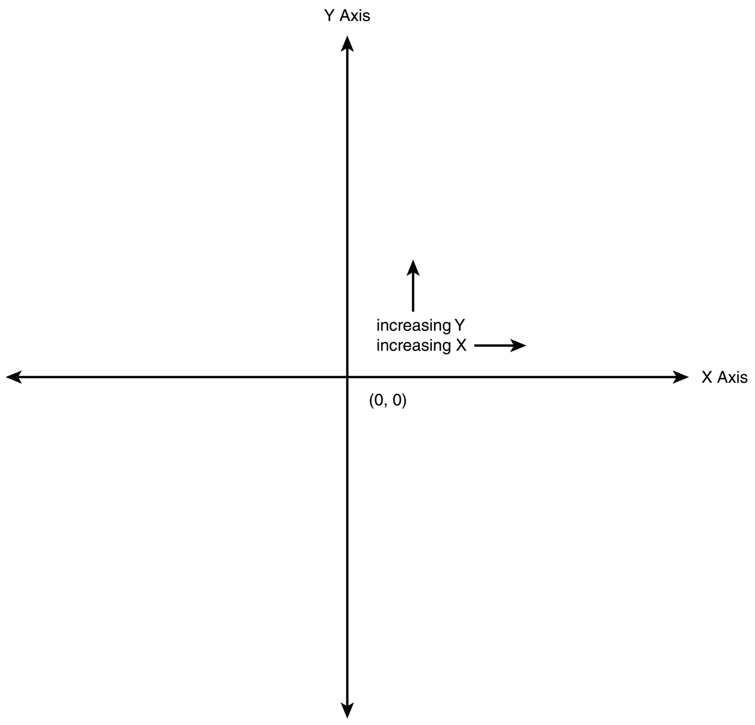
- Understanding the MIDP graphics coordinate system
- Assessing the role of color in MIDP graphics
- Getting to know the `Graphics` class
- Constructing practical graphical MIDlets

The MIDP Graphics Coordinate System

Before you can get started drawing graphics in a MIDlet, you must have a basic understanding of the coordinate system for MIDP graphics. All software graphics systems are designed according to some kind of coordinate system. A coordinate system typically spells out the origin (0,0) of a graphical system, along with the axes and the direction of increasing value for each axis. The traditional mathematical coordinate system familiar to most of us is shown in Figure 7.1.

FIGURE 7.1

The traditional mathematical coordinate system is familiar to most of us as the basis for mathematical graphs.



The MIDP graphics system relies on a coordinate system of its own to specify how and where drawing operations take place. The MIDP coordinate system has its origin located in the upper-left corner of the device screen; positive X values increase to the right, and positive Y values increase down. All values in the MIDP coordinate system are positive integers. Figure 7.2 shows how this coordinate system looks.

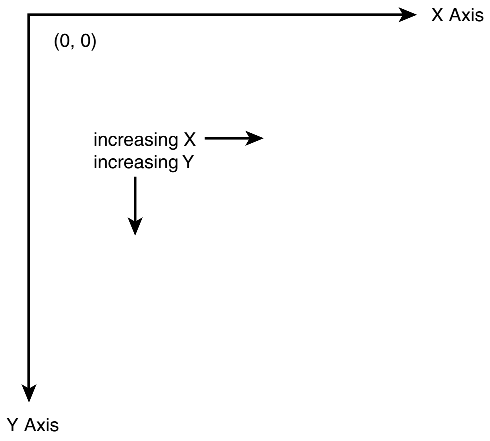


FIGURE 7.2
The MIDP graphics coordinate system has its origin at the upper-left corner of the device screen, with positive X values increasing to the right and positive Y values increasing down.

This coordinate system is very important because every graphical operation in the MIDP API depends on it. One interesting thing to note about the MIDP coordinate system is that it represents locations between pixels, as opposed to pixels themselves. In other words, the pixel in the upper-left corner of the screen has its upper-left corner at (0,0) and its lower-right corner at (1,1). This helps to remove any doubts when filling a primitive graphics shape such as a rectangle—the coordinates specified for the rectangle serve as the bounds of the filled area.

A Quick Color Primer

Although many wireless mobile devices don't have color displays, it's difficult to talk about graphics without mentioning color. As display technologies improve and prices drop, I'm sure more devices will include color displays. This is the same trend that took place in the laptop and notebook computer market several years ago; now it's tough to find a notebook computer without a color screen. You need to understand how color works in the MIDP graphics API to use MIDP graphics fully. Fortunately, color in MIDP graphics is basically a carryover of standard Java color graphics.

Most computer systems take a similar approach to representing color. The main function of color in a computer system is to reflect the physical nature of color accurately within the confines of a graphical system. In Java, this is accomplished by mixing the three primary colors: red, green, and blue. Each pixel on a color display has three color components (red, green, and blue) that combine to form the complete color. The MIDP color system is identical to the standard Java system in that it forms unique colors by using varying intensities of the colors red, green, and blue. More specifically, the combinations of the numeric intensities of the primary colors (red, green, and blue) represent MIDP colors. This color system is known as RGB (Red, Green, Blue) and is standard on most graphical computer systems.

Note

Although RGB is the most popular computer color system in use, there are others. Another popular color system is HSB, which stands for Hue Saturation Brightness. In this system, colors are defined by varying degrees of hue, saturation, and brightness. The HSB color system isn't directly supported in the MIDP graphics API.

To make sure that you understand how the intensities of the primary colors affect the formation of a complete color, Table 7.1 shows the numeric values for the red, green, and blue components of some basic colors.

TABLE 7.1 Numeric Color Component Values for Commonly Used Colors

Color	*Red*	*Green*	*Blue*
White	255	255	255
Black	0	0	0
Light Gray	192	192	192
Dark Gray	128	128	128
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Yellow	255	255	0
Purple	255	0	255

Notice in the table that the intensities of each color component range from 0 to 255 in value. This means that each color is given 8 bits of storage in memory. Also, when the color components are combined to form a complete color, the complete color takes up 24 bits of storage. For this reason, the MIDP color system is referred to as 24-bit color. Of course, that doesn't mean a whole lot for displays that render only shades of gray, but it matters a great deal when it comes to MIDP graphics programming.

It's worth pointing out that the MIDP graphics API doesn't include the familiar `Color` class that is a part of standard Java Advanced Windowing Toolkit (AWT) graphics. Eliminating the `Color` class is part of the streamlining that took place in trying to make the MIDP API as compact as possible. In reality, the `Color` class just served as an organizational structure for the red, green, and blue color components. In MIDP graphics programming, you just reference these color components as individual integers, as opposed to using a `Color` object.

The Graphics Class

If you've done any programming with standard Java, you are no doubt familiar with the `Graphics` class, which provides the majority of functionality for drawing graphics directly within an applet or application. The `Graphics` class provides the capability of drawing graphics primitives (lines, rectangles, and so on), text, and images either to the display or to an offscreen memory buffer. You perform graphics operations by calling methods on a `Graphics` object, which is made available in a MIDlet's `paint()` method. A `Graphics` object is passed into the `paint()` method and is then used to perform graphical output to the MIDlet's screen or an offscreen buffer. Because the `Graphics` object is provided by `paint()`, you never explicitly create a `Graphics` object.

NEW TERM *offscreen buffer*—An area of memory set aside for performing graphics operations, as opposed to performing the operations directly on the display.

Note The `paint()` method is actually part of the `Canvas` class, which represents an abstract drawing surface. To draw graphics using the `Graphics` class, you must create a `Canvas` object and set it as the screen for your MIDlet. You learn how to do this a little later in the lesson.

The `Graphics` class has a few attributes that determine how different graphical operations are carried out. The most important of these attributes is the `color` attribute, which determines the color used in graphics operations such as drawing lines. You set this attribute using the `setColor()` method defined in the `Graphics` class. `setColor()` takes three integer parameters that represent the three primary color components. Similar to `setColor()` is `setGrayScale()`, which takes a single integer parameter that establishes a shade of gray within the range 0 to 255. If a color display is being used, all three-color components are set to the same level, which results in a gray color. Graphics objects also have a `font` attribute that determines the size and appearance of text. This attribute is set using the `setFont()` method, which takes a `Font` object as its only parameter. You learn more about drawing text and using the `Font` object in the section "Drawing Text," later in this lesson.

Note

Another version of the `setColor()` method takes a single integer parameter containing the complete color. The individual red, green, and blue color components are specified within this color value according to the following hexadecimal format: `0x00RRGGBB`. In other words, the red (`RR`), green (`GG`), and blue (`BB`) components are stored as the lower three bytes of the 32-bit integer value.

Most of the graphics operations provided by the `Graphics` class fall into one of the following categories:

- Drawing graphics primitives
- Drawing text
- Drawing images

The next few sections dig a little deeper into these graphics operations and show you exactly how to perform each of them.

Drawing Graphics Primitives

Graphics primitives consist of lines, rectangles, and arcs. You can create pretty impressive graphics by mixing these primitives; the `Graphics` class provides methods for drawing these primitives. You can also use the methods defined in the `Graphics` class to fill the area defined by a primitive with a particular color.

New Term *graphics primitive*—A fundamental graphics shape such as a line, rectangle, oval, or arc.

Lines

A line is the simplest of the graphics primitives and is therefore the easiest to draw. The `drawLine()` method handles drawing lines and is defined as follows:

```
void drawLine(int x1, int y1, int x2, int y2)
```

The first two parameters, `x1` and `y1`, specify the starting point for the line; the `x2` and `y2` parameters specify the ending point. It's important to understand that these coordinates represent the bounds of the starting and ending points for a line. Assuming that you're drawing a line in positive X and Y directions, `x1` and `y1` pinpoint the upper-left corner of the first pixel in the line, and `x2` and `y2` define the lower-right corner of the last pixel in the line. To draw a line in a MIDlet, call `drawLine()` in the MIDlet's `paint()` method, as in this example:

```
public void paint(Graphics g) {
 g.drawLine(5, 10, 15, 55);
}
```

This code results in a line being drawn from the point (5,10) to the point (15,55). Most graphical programming environments provide a means to draw lines (and other graphics primitives) in various widths. The MIDP API doesn't support lines of varying widths, although it is possible to change the style of the line via the `setStrokeStyle()` method. This method accepts one of two constant values, `Graphics.SOLID` and `Graphics.DOTTED`, which determine whether a solid or dotted line is drawn. The `Graphics.SOLID` stroke style is used by default if you don't specify a stroke style.

Note

Unlike the traditional Java graphics API, the MIDP API doesn't include direct support for drawing polygons. Instead, you must draw a polygon by drawing multiple lines with the `drawLine()` method.

Rectangles

Rectangles are also very easy to draw in MIDlets. The `drawRect()` method enables you to draw rectangles by specifying the upper-left corner and the width and height of the rectangle. The `drawRect()` method is defined in the `Graphics` class as follows:

```
void drawRect(int x, int y, int width, int height)
```

The `x` and `y` parameters specify the location of the upper-left corner of the rectangle, and the `width` and `height` parameters specify their namesakes, in pixels. To draw a rectangle using `drawRect()`, just call it from the `paint()` method like this:

```
public void paint(Graphics g) {
 g.drawRect(5, 10, 15, 55);
}
```

This code draws a rectangle that is 15 pixels wide, 55 pixels tall, and has its upper-left corner at the point (5,10). There is also a `drawRoundRect()` method that enables you to draw rectangles with rounded corners:

```
void drawRoundRect(int x, int y, int width, int height, int arcWidth,
 int arcHeight)
```

The `drawRoundRect()` method requires two more parameters than `drawRect()`: `arcWidth` and `arcHeight`. These parameters specify the width and height of the arc forming the rounded corners of the rectangle. If you were to draw an oval that is `arcWidth` wide and `arcHeight` tall and then divide the oval into four sections, each section would represent the rounded corner of a rectangle. Following is an example that uses `drawRoundRect()` to draw a rectangle with rounded corners:

```
public void paint(Graphics g) {
 g.drawRoundRect(5, 10, 15, 55, 6, 12);
}
```

In this example, a rectangle that is 15 pixels wide and 55 pixels tall is drawn at the point (5,10). The corners are rounded and have arcs equivalent to a curved section of an oval that is 6 pixels wide by 12 pixels tall. The `Graphics` class also provides versions of each of these rectangle-drawing methods that fill the interior of a rectangle with the current color, in addition to drawing the rectangle itself. These methods are `fillRect()` and `fillRoundRect()`.

Note To draw a perfect square using either of the rectangle-drawing methods, use an equal width and height.

Arcs

Arcs are more complex than lines and rectangles. An arc is a section of an oval; erase part of an oval and what you have left is an arc. You draw an arc in relation to the complete oval it is a part of. To specify an arc, you must specify a complete oval and what section of the oval the arc comprises. The method for drawing an arc is as follows:

```
void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

The first four parameters of the `drawArc()` method define the oval of which the arc is a part. The remaining two parameters define the arc as a section of this oval. Figure 7.3 shows an arc as a section of an oval.

As you can see in Figure 7.3, an arc is defined within an oval by specifying the starting angle for the arc in degrees, as well as the number of degrees to sweep in a particular direction. The sweep direction is counterclockwise, meaning that positive arc angles are counterclockwise and negative arc angles are clockwise. The arc shown in Figure 7.3 has a starting angle of 95 degrees and an arc angle of 115 degrees. The resulting angle is the sum of these two angles, which is 210 degrees.

Following is an example of drawing an arc using the `drawArc()` method:

```
public void paint(Graphics g) {
 g.drawArc(5, 10, 150, 75, 95, 115);
}
```

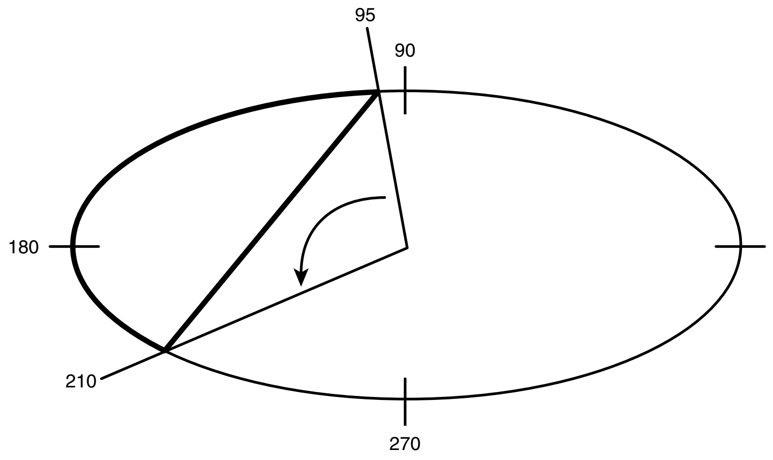


FIGURE 7.3
In MIDP graphics, an arc is described as a section of an oval.

This code draws an arc that is a section of an oval 150 pixels wide and 75 pixels tall, located at the point (5,10). The arc starts at 95 degrees within the oval, and sweeps counterclockwise for 115 degrees.

Unlike standard Java graphics, there is no `drawOval()` method defined in the MIDP `Graphics` class. Instead, you must draw ovals using the `drawArc()` method. To draw an oval, simply specify a sweep angle of 360 degrees as the last parameter to the `drawArc()` method. This means that the arc is to sweep the entire oval, which effectively draws the entire oval.

The method for drawing filled arcs is `fillArc()`. This is very useful because you can use it to draw pie-shaped pieces of a circle or oval. For example, if you are writing an applet to draw a pie graph for a set of statistical data, you can use the `fillArc()` method to draw each piece of the pie.

Note To draw a perfect circle using the `drawArc()` method, use an equal width and height and set the sweep angle of the arc to 360 degrees.

Drawing Text

In Day 8, “Making the Most of MIDlet GUIs,” you learn how to use graphical user interface (GUI) components to do interesting things such as retrieve information from the user and display it in a variety of different ways. While you can use such components to display text, they don’t give you as much freedom as drawing text directly to the display using the `Graphics` class. Fortunately, drawing text with the `Graphics` class is easy and yields good results.

Text is always drawn with the currently selected font. The default font is of a medium size that works well in most situations. However, there will be times when you'll want to use a larger or smaller font or bold or italicize the text. To accomplish this, you first must create a font and select it before drawing any text. The `setFont()` method is used to select a font and is defined as follows:

```
void setFont(Font font)
```

The `Font` object models a textual font and includes the face, style, and size of the font. The `Font` object supports four different font styles, which are represented by the following constant members: `STYLE_PLAIN`, `STYLE_BOLD`, `STYLE_ITALIC`, and `STYLE_UNDERLINED`. These styles are really just constant numbers, and the last three can be used together to yield a combined effect; the `STYLE_PLAIN` style is the absence of the other three styles. Unlike traditional Java graphics programming, you never create a MIDP `Font` object using a constructor. Instead, you call the static `getFont()` method and pass in the face, style, and size parameters as integer constants:

```
static Font getFont(int face, int style, int size)
```

Because fonts are somewhat limited in MIDlets, you must use predefined integer constants for each of these parameters. For example, the font face must be one of the following values: `FACE_SYSTEM`, `FACE_MONOSPACE`, or `FACE_PROPORTIONAL`. Likewise, the style of font must be one of the constants I mentioned earlier: `STYLE_PLAIN` or a combination of `STYLE_BOLD`, `STYLE_ITALIC`, and `STYLE_UNDERLINED`. Finally, the font size is specified as one of the following predefined constants: `SIZE_SMALL`, `SIZE_MEDIUM`, or `SIZE_LARGE`. Following is an example of obtaining a large, bold, underlined font with a monospace face:

```
Font myFont = Font.getFont(Font.MONOSPACE, Font.LARGE,
  Font.BOLD | Font.UNDERLINED);
```

Notice that in MIDP graphics, you don't have the freedom to create fonts of any point size. This makes sense given the limited size of mobile device displays. Also keep in mind that there is no guarantee that a font is available on a given physical device, which is a good example of why it's important to test your MIDlets on a physical device.

Note

If you ever need to obtain the default system font, you can retrieve it by calling the static `getDefaultFont()` method of the `Font` class.

After you've obtained a font using the `getFont()` method, you must select it for drawing text by calling the `setFont()` method, as in the following example:

```
g.setFont();
```

Now you're ready to draw some text using the font you've obtained and selected. The `drawString()` method, defined in the `Graphics` class, is exactly what you need. `drawString()` is defined as follows:

```
void drawString(String str, int x, int y, int anchor)
```

The `drawString()` method takes a `String` object as its first parameter, which contains the text to be drawn. The next two parameters, `x` and `y`, specify the location at which the string is drawn. The specific meaning of this location is determined by the last parameter, `anchor`. To help simplify the positioning of text and images, the MIDP API introduces *anchor points*, which help a great deal in positioning text and images without having to perform any calculations. An anchor point is associated with a horizontal constant and a vertical constant, each of which determines the horizontal and vertical positioning of the text with respect to the anchor point. The horizontal constants used to describe an anchor point are `LEFT`, `HCENTER`, and `RIGHT`. One of these constants is combined with a vertical constant to fully describe an anchor point. The vertical constants are `TOP`, `BASELINE`, and `BOTTOM`.

NEW TERM *anchor point*—An x,y coordinate that is used in conjunction with horizontal and vertical constants to position text and images.

As an example of how to use anchor points, if you want a string of text to appear centered along the top edge of the screen, you might call the `drawString()` method with the following values:

```
g.drawString("Look up here!", getWidth() / 2, 0,
  Graphics.HCENTER | Graphics.TOP);
```

In this code, the x,y position of the text is specified at the top of the display and halfway across (`getWidth() / 2`). Incidentally, I'm assuming this code is placed within a `Canvas`-derived class, which is why I'm able to call the `getWidth()` method to get the size of the display. The meaning of this x,y position is established by the `anchor` parameter, which is specified as the combination of the `Graphics.HCENTER` and `Graphics.TOP` constants. This means that the text is drawn centered horizontally on the x part of the x,y position. It also means that the top of the text is placed at the y part of the x,y position.

In addition to the `drawString()` method, several other methods are used to draw text. The `drawChar()` and `drawChars()` methods are used to draw individual text characters:

```
void drawChar(char character, int x, int y, int anchor)
void drawChars(char[] data, int offset, int length, int x, int y,
  int anchor)
```

Both of these methods work similarly to `drawString()` in that they rely on an x,y position and an anchor to specify precisely where the text is drawn. There is also a `drawSubstring()` method that enables you to draw a substring of text located within a string:

```
void drawSubstring(String str, int offset, int len, int x, int y,
  int anchor)
```

This method includes the additional parameters `offset` and `len` for specifying the substring of text within the string passed as the `str` parameter.

Drawing Images

Images are rectangular graphical objects composed of colored or grayscale pixels. Each pixel in an image describes the color at that particular location of the image. Pixels can have unique colors that are usually described with the RGB color system. Color images in MIDP graphics are 32-bit images, which means that each pixel in an image is described using 32 bits. The red, green, and blue components of a pixel's color are stored in these 32 bits.

NEW TERM *image*—A rectangular graphical objects composed of colored or grayscale pixels.

Before learning the details of how to draw an image, you must first learn how to load images. Because images are typically stored in external files, you must load them from a file before you can draw them. Images are loaded and created using a special static method of the `Image` class called `createImage()`:

```
public static Image createImage(String name) throws IOException
```

To create an image using the `createImage()` method, you specify the name of the image file as the only parameter to `createImage()`:

```
Image img = Image.createImage("Bear.png");
```

The `createImage()` method returns an `Image` object, which can then be used to work with the image within the MIDP graphics API. It is also possible to create a blank `Image` object from scratch by calling a different version of the `createImage()` method that accepts the width and height of the image. The `Image` class represents a graphical image, such as a PNG, GIF, or JPEG file image, and provides a few methods for determining the width and height of the image. `Image` also includes a method for retrieving a `Graphics` object for the image, which enables you to draw directly onto the image.

Note

In case you aren't familiar with it, the Portable Network Graphics (PNG) image format is rapidly gaining in popularity as an improvement over the GIF format. The `Image` class fully supports the PNG image format.

The `Graphics` class provides a single method, `drawImage()`, for drawing images:

```
boolean drawImage(Image img, int x, int y, int anchor)
```

This method probably looks somewhat familiar to you because it uses the same anchor point approach as the `drawString()` method you learned about in the previous section. Similar to `drawString()`, the `drawImage()` method draws the image at the x,y position as determined by the `anchor` parameter. The same horizontal and vertical anchor constants that you learned about earlier apply to images as well.

To summarize, the process of drawing an image involves first calling the static `Image.createImage()` method to create and load the image, followed by a call to `drawImage()`, which actually draws the image on the display. Following is an example of code that loads and draws a centered image within the `paint()` method of a MIDlet:

```
public void paint(Graphics g) {
  // Clear the display
  g.setColor(255, 255, 255);  // White
  g.fillRect(0, 0, getWidth(), getHeight());

  // Create and load the image
  Image img = Image.createImage("Bull.png");

  // Draw the image
  g.drawImage(img, getWidth() / 2, getHeight() / 2,
    Graphics.HCENTER | Graphics.VCENTER);
}
```

In this example, the display is first cleared by setting the color to white and filling the entire display. This is necessary so that you have a clean surface on which to draw. After that, the image `Bull.png` is loaded and created using the `createImage()` method. After the image is created, the `drawImage()` method is used to draw the image centered on the display; the `HCENTER` and `VCENTER` constants are used to specify the anchor point of the image.

Building Your First Graphical MIDlet

You now have a solid understanding of MIDP graphics and are probably anxious to see them at work within the context of a real MIDlet. You learned throughout this lesson that graphics are handled in the `paint()` method of a MIDlet. However, the MIDlet class doesn't include a `paint()` method; instead, you must use the `Canvas` class to perform all graphics operations. The `Canvas` class represents an abstract drawing surface and must be subclassed in a MIDlet to perform any drawing operations using the `Graphics` class. You then establish your derived `Canvas` class as the screen for a MIDlet in order to display it to the user.

To get started with a graphical MIDlet, you must first create a `Canvas`-derived class that is associated with your MIDlet. You then create an object of this class as a member variable of the MIDlet class. The `Canvas` object is then set as the current screen for the MIDlet through a call to the `setCurrent()` method. The best way to understand this process is to work through a simple example. A good way to demonstrate basic MIDP graphics programming is to draw the Olympics symbol, which consists of five interlocking multicolored rings. Let's get started with the `OlympicsCanvas` class, which provides a canvas for the Olympics MIDlet:

```
class OlympicsCanvas extends Canvas {
  public void paint(Graphics g) {
    // Draw the first row of circles
    g.setColor(0, 0, 255);    // Blue
    g.drawArc(5, 5, 25, 25, 0, 360);
    g.setColor(0, 0, 0);      // Black
    g.drawArc(35, 5, 25, 25, 0, 360);
    g.setColor(255, 0, 0);    // Red
    g.drawArc(65, 5, 25, 25, 0, 360);

    // Draw the second row of circles
    g.setColor(255, 255, 0);  // Yellow
    g.drawArc(20, 20, 25, 25, 0, 360);
    g.setColor(0, 255, 0);    // Green
    g.drawArc(50, 20, 25, 25, 0, 360);
  }
}
```

As you can see, this class does nothing more than extend the `Canvas` class and provide a `paint()` method. Within the `paint()` method, the `setColor()` and `drawArc()` methods are used to draw the Olympics symbol. Notice that all of the angle arguments to the `drawArc()` methods are `0` and `360`, which results in complete ovals being drawn.

With the `OlympicsCanvas` class in place, you can declare a member variable of type `OlympicsCanvas` in the `Olympics` MIDlet class:

```
private OlympicsCanvas screen;
```

This member variable must be initialized in the constructor for the `Olympics` class, as the following code demonstrates:

```
screen = new OlympicsCanvas();
```

Now you need to set the canvas as the current screen so that it is made visible to the user. This is easily accomplished in the `startApp()` method, as follows:

```
public void startApp() throws MIDletStateChangeException {
  // Set the current display to the screen
  display.setCurrent(screen);
}
```

One last piece of code is necessary in the Olympics MIDlet that has to do with the fact that you're using graphics features of the MIDP API. This is to import the `javax.microedition.lcdui` package, which houses the MIDP graphics classes and interfaces:

```
import javax.microedition.lcdui.*;
```

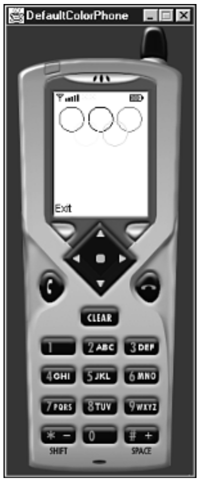
That's really all of the graphics-specific code required in the Olympics MIDlet. Just so you see how it all goes together, following is the complete source code for the Olympics MIDlet:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class Olympics extends MIDlet implements CommandListener {
  private Command exitCommand;
  private Display display;
  private OlympicsCanvas screen;

  public Olympics() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit command
    exitCommand = new Command("Exit", Command.EXIT, 2);

    // Create the main screen form
    screen = new OlympicsCanvas();

    // Set the Exit command
    screen.addCommand(exitCommand);
    screen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the screen
    display.setCurrent(screen);
  }

  public void pauseApp() {
  }
  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
  }
}

class OlympicsCanvas extends Canvas {
  public void paint(Graphics g) {
    // Draw the first row of circles
    g.setColor(0, 0, 255);    // Blue
    g.drawArc(5, 5, 25, 25, 0, 360);
    g.setColor(0, 0, 0);      // Black
    g.drawArc(35, 5, 25, 25, 0, 360);
    g.setColor(255, 0, 0);    // Red
    g.drawArc(65, 5, 25, 25, 0, 360);

    // Draw the second row of circles
    g.setColor(255, 255, 0);  // Yellow
    g.drawArc(20, 20, 25, 25, 0, 360);
    g.setColor(0, 255, 0);    // Green
    g.drawArc(50, 20, 25, 25, 0, 360);
  }
}
```

After compiling, preverifying, and packaging the Olympics MIDlet, you can take it for a spin in the J2ME emulator. Figure 7.4 shows the Olympics MIDlet in all its graphical glory.

FIGURE 7.4

The Olympics MIDlet demonstrates how to draw basic graphics shapes using the MIDP API.



Building a Slide Show MIDlet

Although the Olympics MIDlet example is pretty interesting and definitely helped to get your feet wet with MIDP graphics programming, you're probably still craving a little more. Wouldn't it be nice to see how images and text are drawn within the context of a MIDlet? In this section you work through the details of a slide show MIDlet that serves as a practical example of how to draw images and text.

The SlideShow MIDlet loads several images, along with associated text captions, and displays them one at a time. The user can flip through the "slides" using the left and right arrow keys on the mobile device. Because virtually all of the slide show functionality of the MIDlet is graphical, the majority of the code appears in the `SSCanvas` class, which is derived from `Canvas` and serves as the canvas for displaying a slide. Following are the member variables defined for the `SSCanvas` class, which store the slide images and their respective captions, along with the current slide:

```
private Image[] slides;
private String[] captions = { "Automotive", "Beauty", "Construction",
                              "Pest Control", "Pet Store", "Restaurant" };
private int curSlide = 0;
```

The `slides` variable is an array of `Image` objects that is initialized in the constructor for the `SSCanvas` class. The following is the code for this constructor:

```
public SSCanvas() {
  // Load the slide show images
  try {
    slides = new Image[6];
    slides[0] = Image.createImage("/BizAuto.png");
    slides[1] = Image.createImage("/BizBeauty.png");
    slides[2] = Image.createImage("/BizConstruction.png");
    slides[3] = Image.createImage("/BizPest.png");
    slides[4] = Image.createImage("/BizPet.png");
    slides[5] = Image.createImage("/BizRestaurant.png");
  }
  catch (IOException e) {
    System.err.println("Failed loading images!");
  }
}
```

This code is pretty straightforward in that it creates an `Image` array and then initializes each of its elements by loading an image with the `Image.createImage()` method. It's important to note that each image filename is preceded by a forward slash (`/`), which indicates that it appears in the root directory where the MIDlet is located. This is important because these images are packaged into the JAR file with the MIDlet class and therefore must be readily accessible.

Note

If you're wondering about the slide show images, they represent business types in my board game, "Inc. The Game of Business." Aside from being neat images, I have the rights to them, so it's no problem using them here.

The real work in the `SSCanvas` class takes place in the `paint()` method, which draws the current slide image and caption on the display. The following is the code for the `paint()` method:

```
public void paint(Graphics g) {
  // Clear the display
  g.setColor(255, 255, 255);  // White
  g.fillRect(0, 0, getWidth(), getHeight());

  // Draw the current image
  g.drawImage(slides[curSlide], getWidth() / 2, getHeight() / 2,
    Graphics.HCENTER | Graphics.VCENTER);

  // Set the font for the caption
  Font f = Font.getFont(Font.FACE_PROPORTIONAL, Font.STYLE_BOLD,
    Font.SIZE_MEDIUM);
  g.setFont(f);

  // Draw the current caption
  g.setColor(0, 0, 0);        // Black
  g.drawString(captions[curSlide], getWidth() / 2, 0,
    Graphics.HCENTER | Graphics.TOP);
}
```

The `paint()` method first clears the display so that the remnants of the previous slide are erased. The current slide image is then drawn centered on the display. A bold, medium, proportional font for the text is then obtained for use in drawing the slide caption. Finally, the color is set back to black and the slide caption is drawn centered along the top edge of the display.

With all the painting out of the way, the last chore for the `SSCanvas` class is to process key presses for the left and right arrow buttons, which allow the user to navigate back and forth through the slide show. The following is the code for the `keyPressed()` method, which carries out this chore:

```
public void keyPressed(int keyCode) {
  // Get the game action from the key code
  int action = getGameAction(keyCode);

  // Process the left and right buttons
  switch (action) {
    case LEFT:
      if (--curSlide < 0)
        curSlide = slides.length - 1;
      repaint();
      break;

    case RIGHT:
      if (++curSlide >= slides.length)
        curSlide = 0;
      repaint();
      break;
    }
}
```

The `keyPressed()` method reveals something in MIDP programming that differs significantly from traditional Java programming: game actions. A *game action* is a special key event that is associated with keys typically used in games. The idea is that you can map game actions to different keys and enable the user to customize the user interface for games. In the `keyPressed()` method, the game action associated with the key code is first obtained with a call to `getGameAction()`. The `LEFT` and `RIGHT` constants are then used to check for the Left and Right game actions. If one of these constants produces a match, the current slide is incremented or decremented, and the display is repainted.

That wraps up the `SSCanvas` class, which represents the majority of the work involved in the SlideShow MIDlet. To integrate the canvas with the MIDlet, an instance of the `SSCanvas` class is created as a member variable of the `SlideShow` class:

```
private SSCanvas screen;
```

This variable is then initialized in the `SlideShow` constructor:

```
public SlideShow() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the Exit command
  exitCommand = new Command("Exit", Command.EXIT, 2);

  // Create the main screen form
  screen = new SSCanvas();

  // Set the Exit command
  screen.addCommand(exitCommand);
  screen.setCommandListener(this);
}
```

You now have all the ingredients for the SlideShow MIDlet. To better understand how it all fits together, check out the complete source code for the MIDlet:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;

public class SlideShow extends MIDlet implements CommandListener {
  private Command exitCommand;
  private Display display;
  private SSCanvas screen;

  public SlideShow() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit command
    exitCommand = new Command("Exit", Command.EXIT, 2);

    // Create the main screen form
    screen = new SSCanvas();

    // Set the Exit command
    screen.addCommand(exitCommand);
    screen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the screen
    display.setCurrent(screen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
  }
}

class SSCanvas extends Canvas {
  private Image[] slides;
  private String[] captions = { "Automotive", "Beauty", "Construction",
                                "Pest Control", "Pet Store", "Restaurant" };
  private int curSlide = 0;
```

```
public SSCanvas() {
  // Load the slide show images
  try {
    slides = new Image[6];
    slides[0] = Image.createImage("/BizAuto.png");
    slides[1] = Image.createImage("/BizBeauty.png");
    slides[2] = Image.createImage("/BizConstruction.png");
    slides[3] = Image.createImage("/BizPest.png");
    slides[4] = Image.createImage("/BizPet.png");
    slides[5] = Image.createImage("/BizRestaurant.png");
  }
  catch (IOException e) {
    System.err.println("Failed loading images!");
  }
}

public void keyPressed(int keyCode) {
  // Get the game action from the key code
  int action = getGameAction(keyCode);

  // Process the left and right buttons
  switch (action) {
    case LEFT:
      if (--curSlide < 0)
        curSlide = slides.length - 1;
      repaint();
      break;

    case RIGHT:
      if (++curSlide >= slides.length)
        curSlide = 0;
      repaint();
      break;
    }
}

public void paint(Graphics g) {
  // Clear the display
  g.setColor(255, 255, 255);  // White
  g.fillRect(0, 0, getWidth(), getHeight());

  // Draw the current image
  g.drawImage(slides[curSlide], getWidth() / 2, getHeight() / 2,
    Graphics.HCENTER | Graphics.VCENTER);

  // Set the font for the caption
  Font f = Font.getFont(Font.FACE_PROPORTIONAL, Font.STYLE_BOLD,
    Font.SIZE_MEDIUM);
  g.setFont(f);
```

```
    // Draw the current caption
    g.setColor(0, 0, 0);        // Black
    g.drawString(captions[curSlide], getWidth() / 2, 0,
      Graphics.HCENTER | Graphics.TOP);
  }
}
```

After compiling, preverifying, and packaging the SlideShow MIDlet, you will definitely want to try it out in the J2ME emulator. Don't forget that the left and right arrow buttons are used to navigate back and forth through the slide show. Figure 7.5 shows the SlideShow MIDlet in action.

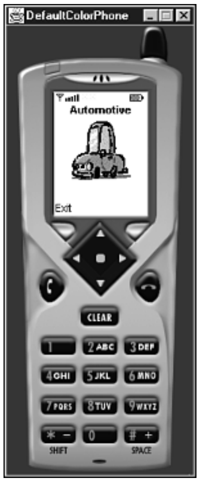


FIGURE 7.5
The SlideShow MIDlet provides an interactive slide show that draws both images and text to the display.

Summary

This lesson bombarded you with much information about graphics support in the MIDP API. Most of it was centered on the `Graphics` and `Canvas` objects, which are very straightforward to use. You began by learning about the MIDP graphics coordinate system and how it affects MIDlet graphics programming. You then moved on to draw a variety of different graphics primitives. From there, you found out how text is drawn through the use of fonts and anchor points. Finally, you concluded the chapter by taking a look at images and how they are drawn. Perhaps the most important aspects of this lesson are the two sample MIDlets that demonstrated most of the graphics concepts you learned.

You're now equipped to create MIDlets that are capable of drawing custom graphics through primitive shapes, text, and images. What you aren't equipped to do is build MIDlets with interesting graphical user interfaces. The next lesson tackles MIDlet GUIs and how they are constructed.

Q&A

Q How does the MIDP graphics system handle color in MIDlets when they are run on a device with a grayscale display?

A If a color MIDlet is run on a device with a grayscale display, the colors are simply converted to shades of gray. It's kind of like a reversal of Ted Turner's colorization process that turns black-and-white movie classics into color movies. Because there are fewer shades of gray than there are colors, it is possible to lose some degree of quality when a color MIDlet runs on a grayscale display.

Q How do I use font faces outside of the three faces that are built into the `Font` class?

A You don't. Don't forget that the MIDP API is designed to be extremely compact, and one facet of this is having limited options for fonts. The three font faces built into the `Font` class should be sufficient in virtually any MIDlet. It would require far too much overhead to support additional fonts in the MIDP graphics system, especially when you consider the fact that it isn't necessary to have that many font options, given the limited display constraints of MIDP devices.

Q How does a MIDlet know where to find an image when it loads it using the `createImage()` method?

A The `createImage()` method looks to the location of the MIDlet class file when attempting to load an image. So it's important to place all of the images for a MIDlet in the JAR file with the MIDlet class when you package the MIDlet for testing and distribution.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. Where is the origin of the MIDP graphics coordinate system located?
2. How is the `Color` object used in MIDP graphics programming?
3. How do you draw a perfect circle using the `Graphics` class?
4. What must you do in order to draw graphics in a MIDlet?

Exercises

1. Using the Olympics MIDlet as a starting point, create a MIDlet that draws a checkerboard on the display. Hint: Use the `fillRect()` method in conjunction with `setColor()` to draw red and black squares.
2. Modify the SlideShow MIDlet so that it uses some of your own images in the slide show. Make sure that you update the captions to match the images.

DAY 8

Making the Most of MIDlet GUIs

Given the limited display constraints of most mobile devices, you might think that the MIDP API doesn't include much support for creating graphical user interfaces (GUIs). In fact, quite the opposite is true. Although not nearly as rich as the standard Java GUI API, the MIDP API still packs a surprising amount of punch when it comes to providing a powerful framework of GUI components for interacting with the user. Granted, it isn't likely that you'll use a mouse to interact with 3D buttons and other fancy GUI components on a mobile device, but I think you'll find the MIDP GUI components quite capable, even with the limited input hardware found on MIDP devices. Similar to the graphics classes you learned about in Day 7, "Building Graphical MIDlets," the GUI classes are located in the `javax.microedition.lcdui` package.

This lesson introduces the various components used to build MIDlet GUIs. If you are accustomed to using standard Java AWT or Swing components, you will definitely notice how the MIDP GUI components are pared down for the mobile environment. Nonetheless, they get the job done quite well, as you will soon find out. Following are the major topics in this lesson:

- Understanding the role of the display
- Using screens and forms as the basis for MIDlet GUIs
- Getting to know the various MIDP GUI components
- Handling MIDlet actions with commands
- Constructing a MIDlet to calculate mortgage payments

Getting to Know the Display

Because all MIDlets are inherently graphical, they all must have some means of being displayed. Knowing this, it is reasonable to consider the display as the most important resource used by a MIDlet. A *display* is an abstract software representation of a mobile device's screen and input controls. Displays are so important that every MIDlet gets its own display to work with. In fact, one of the standard pieces of code you see in every MIDlet is the line that retrieves the `Display` object for the MIDlet, which looks like this:

```
display = Display.getDisplay(this);
```

In this sample code, `display` is a member variable of type `javax.microedition.lcdui.Display`, and it is being set to the display for the MIDlet. You can probably guess from this code that a display is modeled by the `Display` class. The `Display` class is defined in the `javax.microedition.lcdui` package, and includes some important methods that are commonly used in virtually all MIDlets. Before getting into the coding aspects of the `Display` class, however, I'd like to cover a few more conceptual aspects of displays.

NEW TERM *display*—An abstract representation of a mobile device's screen and input controls.

A display provides a representation of the device screen and user input controls for use in a MIDlet. One display is assigned to each MIDlet even though there is only one physical device screen. Even so, only one MIDlet can have access to the physical screen at any given time. In this way, the current display is constantly changing as different MIDlets are launched and exited. In some ways you can think of a MIDlet's association with the current display as similar to input focus in traditional Java applets and applications. Only one applet or application can have the input focus at any given time, which is also true of MIDlets and the current display. This makes sense because any user input should only be made available to a single MIDlet, which is the one that is currently active and visible on the current display.

The visualization of a MIDlet is handled by a *displayable object*, which is of type `Displayable`. A displayable object is a special GUI component that can fill the display with either graphics or user interface components. This is a somewhat vague description of a displayable object, so it might help to consider how the `Displayable` class figures into the MIDP API. There are only two direct subclasses of the `Displayable` class: `Canvas` and `Screen`. As you learned in the preceding lesson, the `Canvas` class represents an abstract drawing surface to which you can draw lines, rectangles, arcs, text, and images. The `Screen` class is a little more interesting because it serves as the base class for several other fundamental GUI components, including the `Form` class. The `Form` class acts as a container that is used to hold other components. The next section delves deeper into the relationship between screens and forms, so for now let's return to the `Display` class.

NEW TERM *displayable object*—A special GUI component that is capable of filling the display with either graphics or user interface components.

It is important to understand that the `Display` class models the physical device display, whereas the `Displayable` class models a GUI component. Every MIDlet has its own instance of the `Display` class that provides access to a device's display and input controls. This object is first made available in the `startApp()` method that is called whenever a MIDlet first starts. The static `Display.getDisplay()` method is used to obtain the `Display` object for a MIDlet, as the following code demonstrates:

```
display = Display.getDisplay(this);
```

This code assumes that you have declared a variable named `display` that is of type `Display`. Additionally, the code must be called within the `startApp()` method or some other method that isn't called until after the MIDlet starts. This `Display` object is valid until the `destroyApp()` method returns, which signals the imminent destruction of the MIDlet. Even if several MIDlets are running and the current MIDlet isn't currently being displayed, its `Display` object is still valid.

To be visible on the display, a MIDlet must create an object that is derived from the `Displayable` class. The `Displayable` class is therefore responsible for rendering a graphical representation of a MIDlet on the display. However, the `Displayable` class is an abstract class, which means that you can't use it directly. Instead, you must use one of the classes derived from `Displayable` such as `Canvas` or `Screen`. Regardless of what type of `Displayable`-derived class you use to construct the user interface for a MIDlet, you must call a special method on the `Display` object for a `Displayable`-derived class to be made visible to the user. This is the `setCurrent()` method, which has the following form:

```
void setCurrent(Displayable next)
```

Note

Another version of the `setCurrent()` method displays an alert before displaying the specified `Displayable` object. You learn about alerts later in the lesson in the section titled "Assessing the MIDP GUI Components."

This method is rather simple in that it accepts a `Displayable` object as its only parameter. You will use this method any time it is necessary to display a screen or canvas within a MIDlet, which is quite often. If for some reason you need to obtain the current `Displayable` object, you can call the `getCurrent()` method, which has the following form:

```
Displayable getCurrent()
```

In addition to the `getCurrent()` and `setCurrent()` methods used to control the currently displayed screen, canvas, or other GUI component, the `Display` class also includes a couple of methods for finding out information about the device's display capabilities. More specifically, the `isColor()` and `numColors()` methods retrieve information about the color capabilities of the device. Following are the prototypes of these methods, which should give you a pretty clear idea as to how the methods are used:

```
boolean isColor()
int numColors()
```

Working with Screens and Forms

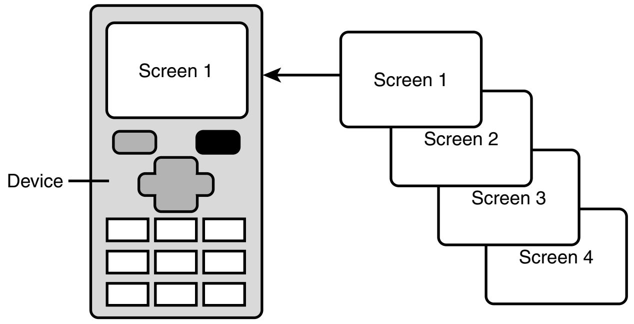
The `Displayable` class has two direct descendents: `Canvas` and `Screen`. You learned in the previous lesson that the `Canvas` class supports direct graphics operations such as drawing lines, rectangles, arcs, text, and images. The `Canvas` class is designed to be shown by itself on the display and therefore doesn't really concern itself with other GUI components. The `Screen` class, on the other hand, is designed to represent a screen within the MIDlet's user interface. A screen is a general GUI component designed to fill the display and serve as a functionally discrete part of a MIDlet's user interface. Although a MIDlet is capable of utilizing only one screen, in most cases several screens are required to fully implement a MIDlet's user interface.

NEW TERM *screen*—A general GUI component designed to fill the display and serve as a functionally discrete part of a MIDlet's user interface.

If this description of a screen is a little difficult to swallow, maybe a better explanation is in order. Because MIDlets don't have the luxury of full-size displays on which to construct user interfaces that take up lots of screen real estate, they must operate more efficiently. Part of this efficiency enters into the picture with a MIDlet's user interface, which is organized into screens. A typical MIDlet has several screens that perform different functions such as gathering data from the user and then displaying the results. Because only one screen is visible at a time, most MIDlets are in a constant state of flux as they change screens to accommodate the user. In this way, you can think of MIDlet screens as similar to a stack of note cards; only the top card is visible at any given time, but all of the cards serve a purpose and are capable of being called to the top when necessary. Figure 8.1 shows how screens are logically equivalent to a stack of note cards.

FIGURE 8.1

MIDlet screens are similar to a stack of note cards in that only one screen is visible on the device display at any given time.



Basic MIDlet screens are modeled by the `Screen` class, which is an abstract class. The `Screen` class provides basic functionality for a screen, which primarily consists of a title that appears along the top of the screen when it is displayed. You can get and set the title of a screen using the `getTitle()` and `setTitle()` methods, which have the following form:

```
String getTitle()
void setTitle(String s)
```

In addition to the title attribute, screens are also capable of containing a ticker, a line of scrolling text that appears across the screen similar to ticker tape. The ticker for a screen is actually of type `Ticker`, which is a GUI component class that you learn about later in the lesson. To get or set the ticker for a screen, call the `getTicker()` and `setTicker()` methods, which look like this:

```
Ticker getTicker()
void setTicker(Ticker ticker)
```

The `Screen` class is abstract, so you're probably guessing that some useful classes are derived from `Screen`. In fact, four classes that provide surprisingly different features are derived from `Screen`:

- **`Form`**—Serves as a container for building user interfaces with other child components

- **`Alert`**—Displays a text message and an optional image to the user
- **`List`**—Displays a list of items for user selection
- **`TextBox`**—Provides a simple text editor

All of these classes represent GUI components that are designed to fill the entire display, which is why they are derived from the `Screen` class. Of these classes, you might notice that the `Form` class stands apart as playing a unique role of serving as a container for other components. If you have any HTML development experience, you are no doubt familiar with forms and how they are used to gather input from the user. Just as an HTML form consists of several GUI components to form a complete user interface, so does the `Form` class. In other words, the `Form` class is designed to enable you to create child components and build custom user interfaces for your MIDlets. Keep in mind, however, that a form is but one type of MIDlet screen, so you can still use the other screen-derived classes to carry out other functions.

NEW TERM *form*—A GUI component derived from the Screen class that acts as a container for holding other GUI components. Forms are extremely useful in creating highly customized user interfaces.

A form is a container that is capable of holding other GUI components. Only GUI components that are derived from the `Item` class can be placed on a form. The `Item` class models a component that isn't designed to fill the entire display. It is expected that multiple items will appear together on the display at once, thereby forming a complete user interface. You design a custom user interface by creating a form and then adding items to it. To add an item to a form, you call the `append()` method on the form and pass in the item. Following is the prototype for the `append()` method:

```
int append(Item item)
```

Each item added to a form is assigned an index within the form, starting at zero. This index is returned by the `append()` method when you add an item to the form. You can then reference items within a form using the index. The significance of form indexes is that they determine the order in which items are displayed on the form. It is therefore important to be able to manipulate items within a form through their indexes. For example, to delete an item from a form you use the `delete()` method, which accepts an integer index as its only parameter:

```
void delete(int index)
```

You can also insert an item at a specified index in a form using the `insert()` method:

```
void insert(int index, Item item)
```

If you'd rather replace an item on a form with a different item instead of just inserting the item, you can use the `set()` method:

```
void set(int index, Item item)
```

If you'd like to get a specific item from a form, you can do so with the `get()` method:

```
Item get(int index)
```

Finally, you might have a situation where you want to determine the total number of items in a form. You can retrieve this information with a simple call to the `size()` method:

```
int size()
```

To put a few of these methods into perspective, assume that you want to dynamically clear a form of all its items. To accomplish this, you would need a loop that iterated through each item, removing it from the form. Following is the code that accomplishes this task for a form named `form`:

```
for (int i = form.size() - 1; i >= 0; i--)
  form.delete(i);
```

One other important issue related to forms and items is the detection of items changing state. When the user interacts with an item on a form and changes its value, you might want to know about it and take some kind of action. You can do this by creating an item state listener and responding to the item state change. The `ItemStateListener` interface describes a single method, `itemStateChanged()`, that is used to respond to item state changes. Following is the prototype for this method:

```
void itemStateChanged(Item item)
```

To respond to an item state change, you must implement the `ItemStateListener` interface in your MIDlet class and provide an implementation of the `itemStateChanged()` method. You must also register the MIDlet class using the `setItemStateListener()` method, which is defined in the `Form` class. Following is an example of how you would set the current MIDlet as an item state listener for a form using this method:

```
form.setItemStateListener(this);
```

Getting back to the `Item` class, you now know that a GUI component must be derived from the `Item` class to be used within a form. Several classes within the MIDP API are derived from the `Item` class, which give you lots of flexibility in designing and creating MIDlet forms. The next section introduces you to these different components.

Assessing the MIDP GUI Components

All of the GUI components supported by the MIDP API are included in the `javax.microedition.lcdui` package. You're already familiar with the `Canvas`, `Screen`, and `Form` classes, so let's look at the other GUI components that come into play in MIDlet user interface construction. Perhaps the best place to start is to cover the three other
component classes that are derived from the `Screen` class: `Alert`, `List`, and `TextBox`. These three classes are derived from the `Screen` class because they each take up the entire display. This means that they must be displayed individually as screens, and therefore cannot be added to forms. You get into the details of these classes in a moment.

Another interesting GUI component that impacts screens is the `Ticker` class, which is somewhat of an anomaly in that it doesn't derive from any other MIDP GUI class. The `Ticker` class is designed for use with any screen-derived class, and implements a scrolling text window across the top of the screen that is similar to ticker tape. You could use a ticker within a MIDlet screen to display a live data feed such as stock quotes or weather warning information. Tickers are unique within the GUI framework of MIDlets because they are built into the `Screen` class; every `Screen`-derived object is capable of having a ticker along its top edge. The `Ticker` class is covered in more detail in just a moment.

In addition to the `Alert`, `List`, `TextBox`, and `Ticker` classes, the following GUI components are included in the MIDP API:

- `StringItem`
- `ImageItem`
- `TextField`
- `DateField`
- `ChoiceGroup`
- `Gauge`

All of these classes are derived from the `Item` class, which means that they can be used within a form. These classes form the building blocks of custom GUI design in MIDlets. Of course, you will also probably find yourself using the `Alert`, `List`, `TextBox`, and `Ticker` classes to round out the complete user interfaces for your MIDlets, but the `Item`-derived classes serve as the basis for creating forms.

The next few sections explore all of the MIDP GUI components in more detail.

The `Alert` Class

The `Alert` class is derived from `Screen`, and implements a screen that displays information to the user for a specified amount of time, or until the user selects a certain command such as `OK`. An alert is a MIDP equivalent of a message box in traditional Java programming. Alerts are commonly used to display error messages or other pertinent information to the user. To create an alert, you must use the `Alert()` constructor and pass in a few parameters describing the alert:

```
Alert(String title, String alertText, Image alertImage, AlertType alertType)
```

The title of an alert is text that appears across the top of the screen, while the alert text serves as the body text for the alert message. You can also display an image within an alert through the `alertImage` parameter. And finally, and perhaps most importantly, the `alertType` parameter is used to describe the type of the alert. Alert types are described in the `AlertType` class, which defines several embedded `AlertType` objects that represent the different alert types. Following is a list of these built-in objects:

- `ALARM`
- `CONFIRMATION`
- `ERROR`
- `INFO`
- `WARNING`

Note

If you don't want to display an image within an alert, you can pass null for the `alertImage` parameter in the `Alert()` constructor.

The names of these `AlertType` objects are relatively self-explanatory because they describe the function of an alert. Practically speaking, the type of an alert primarily impacts the sound that is played when the alert is displayed. For example, a different tone is played for a `WARNING` alert than for an `INFO` alert. The alert types are primarily used in conjunction with the `Alert` class but they can also be used by themselves if you're only interested in playing an alert sound. For example, the following code plays an error alert sound:

```
AlertType.ERROR.playSound(display);
```

This code uses the built-in `AlertType.ERROR` object to call the `playSound()` method and play the error alert sound. Because the output of sound ultimately must be handled by the display, the `playSound()` method requires a `Display` object as its only parameter.

In the `Alert` class, you now know that you set the type of an alert using an `AlertType` object. Following is an example of how you might create an alert that notifies the user of low device memory:

```
Alert alert = new Alert("Memory Warning",
  "Device memory is running very low.", null,
  AlertType.WARNING);
```

This code creates a warning alert without an image. By default alerts will only be displayed for a second or two, and then will automatically disappear. The specific value of this default timeout period varies across MIDP implementations, so I can't give you an exact value for it. However, if you want to provide an exact value you can do so by calling the `setTimeout()` method, which accepts a timeout period in milliseconds:

```
void setTimeout(int time)
```

It might be that you want the alert to be modal, which means that instead of being displayed for a given period of time the user of the alert must select a command to make it go away. In this case, you can pass the constant `Alert.FOREVER` to the `setTimeout()` method. If you use the `Alert.FOREVER` constant to create a modal alert, a `Done` command will automatically be created for the alert, which allows the user to exit it. Following is an example of how you would use this constant to create a modal alert:

```
alert.setTimeout(Alert.FOREVER);
```

To actually display an alert to the user, you must set it as the current screen using the `setCurrent()` method of the `Display` class. Following is an example of how you would display the "Memory Warning" alert:

```
display.setCurrent(alert);
```

To give you a better idea as to how an alert appears on a device display, Figure 8.2 shows the "Memory Warning" alert.

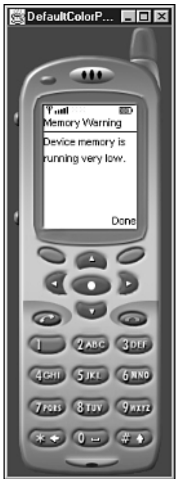
The `List` Class

Similar to `Alert`, the `List` class is also derived from the `Screen` class but serves a much different function than `Alert`. The `List` class provides a screen that contains a list of choices from which the user can select. The `List` class implements the `Choice` interface, which defines constants that describe three basic types of choice lists:

- **`EXCLUSIVE`**—A list that enables only one element to be selected at any given time
- **`IMPLICIT`**—An exclusive list where the currently focused element is selected in response to a command
- **`MULTIPLE`**—A list that allows one or more elements to be selected at any given time

FIGURE 8.2

The Alert component enables you to display important information to the user in a temporary window.



Device memory is running very low.

These constants enter the picture in the `List` class when you create a list using one of the `List()` constructors:

```
List(String title, int listType)
List(String title, int listType, String[] stringElements,
  Image[] imageElements)
```

The first `List()` constructor accepts a title for the list along with the list type. The second constructor goes a step further by enabling you to initialize the list with elements. The string and image arrays passed into this constructor are used as the basis for initializing the list. In other words, the size of these arrays determines how many elements are added to the list. When a list is created using one of the `List()` constructors, you can set it as the current screen in a MIDlet through the `setCurrent()` method. Following is an example of how you might create a list of cities using the `List` class:

```
String[] cities = { "Nashville", "Knoxville", "Memphis", "Chattanooga" };
List list = new List("Select a City", List.EXCLUSIVE, cities, null);
```

Figure 8.3 shows the end result of creating a city list using this sample code.

After creating a list, you can begin interacting with the list using several methods in the `List` class that are implemented from the `Choice` interface. To get the selected element in an exclusive or implicit list, the `getSelectedIndex()` method returns the zero-based index of the currently selected item in the list:

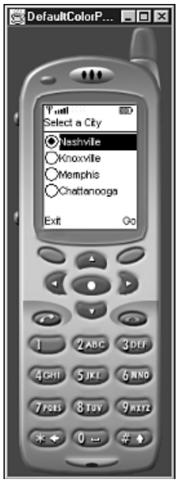
```
int getSelectedIndex()
```

For multiple lists, you must use the `getSelectedFlags()` method, which returns an array that is the size of the number of elements in the list. The `getSelectedFlags()` method returns an array of Boolean values that have a one-to-one relationship with the elements in the list. For each element in the list, a corresponding Boolean value in the array is set to `true` if the element is selected or `false` if it is not. The prototype for the `getSelectedFlags()` method is as follows:

```
int getSelectedFlags(boolean[] selectedArray)
```

FIGURE 8.3

The List component enables you to create a list of elements from which the user can select.



You can also set the selected item or items within a list through the `setSelectedIndex()` and `setSelectedFlags()` methods:

```
void setSelectedIndex(int index, boolean selected)
void setSelectedFlags(boolean[] selectedArray)
```

The index of an element in the list is the key to obtaining its value and determining its state. For example, the `getString()` method takes an index parameter and returns the string value of the element. Similarly, you can determine the selection state of an item by calling the `isSelected()` method and passing in the index.

The `List` class also includes several methods for manipulating the list by adding, inserting, and deleting items. The `append()` method is used to add an element to the end of the list. The `insert()` method expects an index, and inserts an element at the specified index in the list. To replace an element, you can specify an index in the `set()` method as well as the element to use as the replacement. And finally, the `delete()` method expects an index, and enables you to delete the specified element. In addition to adding, replacing, and removing elements, you can determine the number of elements in a list by calling the `size()` method.

The `TextBox` Class

The `TextBox` class is the last of the `Screen`-derived classes tackled in this lesson, and implements a full-screen text-editing component. The `TextBox` class is relatively straightforward to use, and includes a single constructor that enables you to create a text box with a title, default text, a maximum size, and a few constraints:

```
TextBox(String title, String text, int maxSize, int constraints)
```

The `title` parameter of the `TextBox()` constructor appears along the top of the display, while the `text` parameter is used to initialize the text within the box, if necessary. The `maxSize` parameter specifies the maximum number of characters of text that can be entered into the text box. Perhaps more interesting is the `constraints` parameter, which describes any constraints that apply to the entry of text in the box. The `constraints` parameter is specified as one of a number of constants that are defined in the `TextField` class. Following are these constants and their meanings:

- **`ANY`**—There are no constraints on the text.
- **`EMAILADDR`**—The user is only allowed to enter an e-mail address.
- **`NUMERIC`**—The user is only allowed to enter a numeric integer value.
- **`PASSWORD`**—The text is masked so that the typed characters aren't visible.
- **`PHONENUMBER`**—The user is only allowed to enter a phone number.
- **`URL`**—The user is only allowed to enter a network URL.

The `ANY` constraint constant is the most flexible in that it doesn't impose any constraints on the text in the text box. The remaining constraint constants aid in the entry of specific types of information such as e-mail addresses, passwords, and phone numbers. Following is an example of how to use the `ANY` constraint constant to create a text box that might be used with a Diary MIDlet:

```
TextBox box = new TextBox("My Diary", "Dear Self,", 256, TextField.ANY);
```

Figure 8.4 shows the end result of creating a text box using this sample code.

After the text box is created, it is possible to interact with the text within it through several methods. To retrieve the text within the text box as a string, you can call the `getString()` method, which is defined as follows:

```
String getString()
```

Optionally, you can get the text as a character array by calling the `getChars()` method, which looks like this:

```
int getChars(char[] data)
```

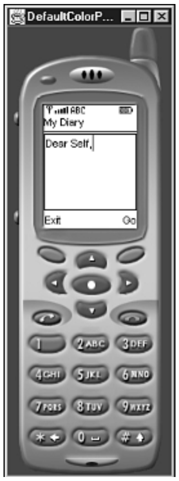


FIGURE 8.4
The TextBox component enables you to create a full-screen text editor to allow the user to enter text.

This method copies the text in the text box into the specified character array, and returns the total number of characters copied. If you prefer to set the contents of the text box, you can do so using the `setString()` and `setChars()` methods, which are as follows:

```
void setString(String text)
void setChars(char[] data, int offset, int length)
```

It is also possible to insert text into a text box by calling one of the `insert()` methods. Following are the two different versions of the `insert()` method, which operate on a string and a character array, respectively:

```
void insert(String src, int position)
void insert(char[] data, int offset, int length, int position)
```

The `position` parameter in these two methods specifies the position that is within the text box contents where you want to insert the new text.

You might find a few other methods in the `TextBox` class useful for manipulating the text stored within a text box. The `delete()` method enables you to delete a selection of text from a text box and the `size()` method enables you to determine how many characters of text are currently stored in the box. You can also find out the maximum number of characters capable of being stored in the box by calling the `getMaxSize()` method.

The `Ticker` Class

The `Ticker` class represents a ticker tape component used to display a scrolling line of text on the display. The `Ticker` class is unique in the MIDP GUI classes because it doesn't derive from either the `Screen` class or the `Item` class. The `Ticker` class is designed for use exclusively within a screen. More specifically, the `Screen` class accommodates a ticker along the top of the display. To create a ticker, use the `Ticker()` constructor and pass in the text that is to be scrolled across as the ticker text. Following is the prototype for this constructor:

```
Ticker(String str)
```

To use a ticker with a screen, you must first create the ticker using the `Ticker()` constructor and then set it for the screen with a call to the `setTicker()` method. Following is an example of how you might add a ticker to the text box code you saw in the previous section:

```
Ticker ticker = new Ticker(Calendar.getInstance().toString());
TextBox box = new TextBox("My Diary", "Dear Self,", 256, TextField.ANY);
box.setTicker(ticker);
```

The ticker created in this code is shown in Figure 8.5.

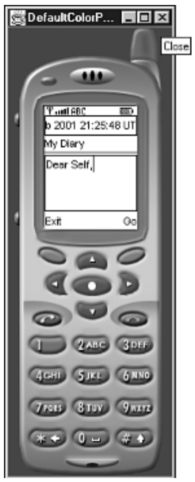


FIGURE 8.5
The Ticker component enables you to display scrolling text along the top of a screen.

In some dynamic MIDlets it might make sense to alter the text being displayed in a ticker. It is possible to get and set the text within a ticker by calling the `getString()` and `setString()` methods, respectively. The prototypes for these methods follow:

```
String getString()
void setString(String str)
```

The `StringItem` Class

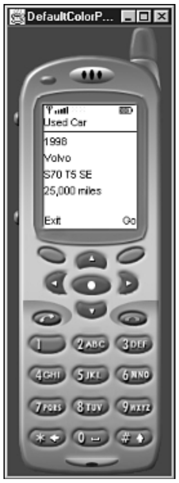
The `StringItem` class is the first of several classes that are all derived from the `Item` class. The `Item` class is significant because it provides the necessary support required to display a component within a form. The `StringItem` class represents a simple item that contains a string of text. This class is used primarily to display text to the user on a form. A string item is described by two pieces of information—a label and text content. Although the text content of a string item is stored and displayed separately from the label within the component, the user is not able to edit it. Following is the constructor for the `StringItem` class:

```
StringItem(String label, String text)
```

To create a string item, use this constructor and pass in a label and text. If you only want to display a single piece of text in a string item, you can leave the `label` parameter as an empty string (`""`).The following example illustrates how one might create several string items and add them to a form:

```
Form form = new Form("Used Car");
StringItem yearItem = new StringItem("", "1998");
StringItem makeItem = new StringItem("", "Volvo");
StringItem modelItem = new StringItem("", "S70 T5 SE");
StringItem milesItem = new StringItem("", "25,000 miles");
form.append(yearItem);
form.append(makeItem);
form.append(modelItem);
form.append(milesItem);
```

Figure 8.6 shows how these string items appear within the form.



```
Used Car
1998
Volvo
S70 T5 SE
25,000 miles
```

FIGURE 8.6

The StringItem component is used within a form to display text.

The `StringItem` class only includes a couple of methods that are used to get and set the text in the string item. Following are the prototypes for these methods:

```
String getText()
void setText(String text)
```

The `ImageItem` Class

The `ImageItem` class is similar to the `StringItem` class except that it is designed to display images instead of text. Image items are created using the `ImageItem()` constructor, which has the following form:

```
ImageItem(String label, Image img, int layout, String altText)
```

As the constructor reveals, every string item has a label, which you can specify as an empty string if you don't want a label displayed with the image. The image itself is passed as the second parameter to the constructor, and is of type `Image`. A `layout` parameter determines how the image item is positioned on the form relative to other items. And finally, the `altText` parameter is used to specify text that is displayed in lieu of the image.

The layout of an image is determined by integer constants defined in the `ImageItem` class, which are as follows:

- **`LAYOUT_DEFAULT`**—Positions the image using the default formatting of the form
- **`LAYOUT_LEFT`**—Aligns the image along the left edge of the form
- **`LAYOUT_RIGHT`**—Aligns the image along the right edge of the form
- **`LAYOUT_CENTER`**—Centers the image horizontally
- **`LAYOUT_NEWLINE_BEFORE`**—Starts a new line for the image
- **`LAYOUT_NEWLINE_AFTER`**—Starts a new line of items after the image

To understand how these layout constants work, understand that items on a form are displayed one line after another moving down the display. The `LAYOUT_DEFAULT` constant cannot be used with any of the other constants, and indicates that an image is to be laid out just as if it were any other item being positioned on the form. The `LAYOUT_LEFT`, `LAYOUT_RIGHT`, and `LAYOUT_CENTER` constants dictate how an image is arranged horizontally on a form. One of the horizontal constants can be combined with the `LAYOUT_NEWLINE_BEFORE` and `LAYOUT_NEWLINE_AFTER` to determine exactly how an image item is positioned on a form. The `LAYOUT_NEWLINE_BEFORE` constant results in an image item being displayed on a new line, below the previous item in the form. Likewise, the `LAYOUT_NEWLINE_AFTER` constant indicates that the next item on the form is to appear on a new line.

Note

You will usually use one of the horizontal layout constants in conjunction with both vertical constants when specifying the layout of an image item.

Before you can create an image item, you must first create the image that you want displayed. The static `createImage()` method of the `Image` class handles this task with ease. The following example demonstrates how one might load an image for use with the used car example in the previous section:

```
Image logoImage = null;
try {
  logoImage = Image.createImage("/Volvo.png");
}
catch (IOException e) {
  System.err.println("EXCEPTION: " + e);
}
```

Note

Images used with the `ImageItem` class must be immutable, which means that they cannot change after they are created. The `Image` class supports the creation of both mutable and immutable images, but any time you create an image from an image file it is automatically immutable.

In this code, the image `Volvo.png` is loaded and created as an `Image` object thanks to the `createImage()` method. It is necessary to handle an `IOException` because the `createImage()` method is defined as being capable of throwing such an exception. Now that you have an image, you can go ahead and create an image item using it. Following is an example of how to create an image item using the `ImageItem()` constructor and incorporate it into the used car string item example from the previous section:

```
form = new Form("Used Car");
ImageItem logoItem = new ImageItem("", logoImage,
  ImageItem.LAYOUT_DEFAULT, "Volvo");
StringItem yearItem = new StringItem("", "1998");
StringItem makeItem = new StringItem("", "Volvo");
StringItem modelItem = new StringItem("", "S70 T5 SE");
StringItem milesItem = new StringItem("", "25,000 miles");
form.append(logoItem);
form.append(yearItem);
form.append(makeItem);
form.append(modelItem);
form.append(milesItem);
```

Notice in this code that the `LAYOUT_DEFAULT` layout constant is used to position the image item. Figure 8.7 reveals how this layout impacts the string items on the form.

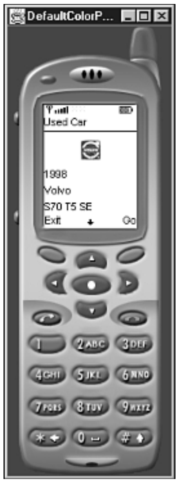


FIGURE 8.7

The ImageItem component enables you to display an image within a form.

The `TextField` Class

The `TextField` class offers a text editor that is designed for use within forms. This differentiates the class from the `TextBox` class, which is an actual screen, and cannot be used in a form alongside other components. However, the `TextField` class is similar in many ways to the `TextBox` class. In fact, you interact with a text field using the exact methods you learned about earlier in the lesson when you covered text boxes. The constructor for the `TextField` class even brings back memories of the `TextBox` class:

```
TextField(String label, String text, int maxSize, int constraints)
```

The `label` parameter of the `TextField()` constructor establishes the label for the component, while the `text` parameter is used to initialize the text within the text field, if desired. The `maxSize` parameter specifies the maximum number of characters of text that can be entered into the text field. The `constraints` parameter, which you hopefully have at least a faint memory of from the `TextBox()` constructor, describes any constraints that apply to the entry of text in the box. If you recall from earlier in the lesson, the `TextField` class actually defines the constraint constants used by both text fields and text boxes. At the risk of being repetitive, take a look at these constants one more time:

- **`ANY`**—There are no constraints on the text.
- **`EMAILADDR`**—The user is only allowed to enter an e-mail address.
- **`NUMERIC`**—The user is only allowed to enter a numeric integer value.

- **`PASSWORD`**—The text is masked so that the typed characters aren't visible.
- **`PHONENUMBER`**—The user is only allowed to enter a phone number.
- **`URL`**—The user is only allowed to enter a network URL.

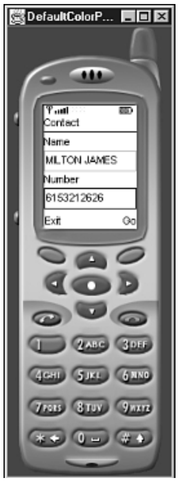
Following is an example of how to create a couple of text fields that might be used with a Contacts MIDlet:

```
form = new Form("Contact");
TextField nameField = new TextField("Name", "", 40, TextField.ANY);
TextField phoneField = new TextField("Number", "", 10, TextField.PHONENUMBER);
form.append(nameField);
form.append(phoneField);
```

Figure 8.8 shows how these text fields enable you to enter the name and phone number for a contact.

FIGURE 8.8

The TextField component.



Contact
Name
MILTON JAMES
Number
6153212626
Exit Go

The methods that enable you to manipulate a text field are literally identical to those used with the `TextBox` class. For this reason, and for the sake of brevity, I encourage you to refer to the discussion of the `TextBox` class earlier in this lesson if you'd like to know more about how to manipulate the text in a text field.

The `DateField` Class

The `DateField` class is fairly unique among the MIDP component items in that it provides an intuitive GUI for entering dates and times. To create a date field, you use the `DateField()` constructor, which has the following form:

```
DateField(String label, int mode)
```

Note

Another version of the `DateField()` constructor that enables you to specify a third parameter, which is the time zone of the date. If you use the two-parameter constructor and don't specify a time zone, the time zone for the local device environment will automatically be used.

This constructor requires the label of the date field, which is standard for all item components. The `mode` parameter to the `DateField()` constructor is much more interesting because it describes in what mode the date field is to operate. Date field modes are defined as constants within the `DateField` class:

- **`DATE`**—Input date information only (day, month, and year)
- **`TIME`**—Input time information only (hours and minutes)
- **`DATE_TIME`**—Input both date and time information

These constants reveal that a date field is capable of being used to input a date, a time, or both a date and time. Following is an example of how you could use the `DateField` class to enable the user to enter a birthday for a contact:

```
form = new Form("Contact");
TextField nameField = new TextField("Name", "", 40, TextField.ANY);
TextField phoneField = new TextField("Number", "", 10, TextField.PHONENUMBER);
DateField bdayField = new DateField("Birthday", DateField.DATE);
form.append(nameField);
form.append(phoneField);
form.append(bdayField);
```

In this code the `DATE` constant is used to set the mode of the date field so that the user can only enter a date. Figure 8.9 shows how this field looks when it first comes into view in the J2ME emulator.

To actually enter a date using the DateField component, you must click the action button on the device to invoke the date selection GUI. Figure 8.10 shows how the date selection GUI presents a highly intuitive approach to selecting a date.

After selecting a date using the DateField component's date selection GUI, the date is displayed in the date field on the form, as shown in Figure 8.11.

The `DateField` class includes `getDate()` and `setDate()` methods to enable you to get and set the date/time that is stored in a date field. Following are the prototypes for these two methods:

```
Date getDate()
void setDate(Date date)
```

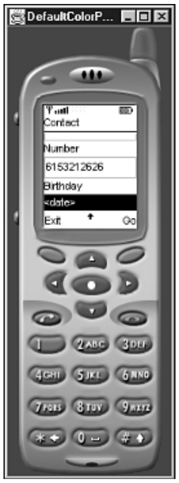


FIGURE 8.9
The DateField component provides a GUI for entering dates and times.

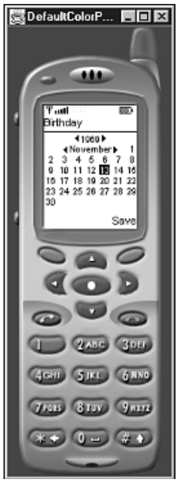


FIGURE 8.10
The DateField component includes a date selection GUI that makes it very intuitive to select a date.

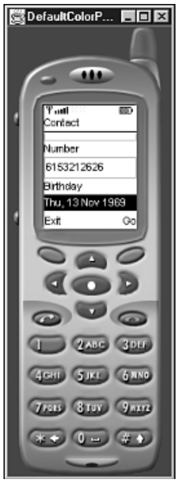
The `ChoiceGroup` Class

The `ChoiceGroup` class provides a means of presenting a list of elements from which the user can select. This class is very similar to the `List` class, except that `ChoiceGroup` is designed as an item component that can be used in forms. Beyond that, however, there isn't much of a difference between the two classes from a coding perspective. For example, take a look at the constructors for the `ChoiceGroup` class:

```
ChoiceGroup(String label, int choiceType)
ChoiceGroup(String label, int choiceType, String[] stringElements,
  Image[] imageElements)
```

FIGURE 8.11

The DateField component.



Contact

Number

6153212626

Birthday

Thu, 13 Nov 1969

Exit Go

The first `ChoiceGroup()` constructor accepts a label for the choice group along with the type. The second constructor enables you to initialize the elements within the choice group using a string array and an image array. These constructors work exactly like their counterparts for the `List` class, which you learned about earlier in the lesson. In fact, you can take the `List` example code from earlier and change the class to `ChoiceGroup` for it to function as a choice group within a form, like this:

```
form = new Form("Tennessee Cities");
String[] cities = { "Nashville", "Knoxville", "Memphis", "Chattanooga" };
ChoiceGroup cityChoice = new ChoiceGroup("Select a City", List.EXCLUSIVE,
➥cities, null);
form.append(cityChoice);
```

Figure 8.12 shows the modified list example code functioning as a choice group within a form.

As Figure 8.12 shows, a choice group appears virtually identical to a list, except that it is within a form that can house additional components. The `ChoiceGroup` class supports all of the same methods as the `List` class for manipulating the elements in the group. Refer to the `List` class discussion for more details regarding these methods.

The `Gauge` Class

The last component in this whirlwind tour of MIDP GUI components is the `Gauge` class, which represents another item component that can be used within forms. The `Gauge` class implements a bar graph that can be used to visually display a value within a range or a percentage completion of a task. A gauge has a maximum value, which establishes its range (0 to maximum), as well as a current value that determines where the gauge is positioned. One interesting aspect of the `Gauge` class is that it can function either interactively or non-interactively. A non-interactive gauge is what you might have expected of the `Gauge` class in that it displays a bar graph. An interactive gauge takes things a step farther by enabling the user to alter the bar graph and use it as an input component.

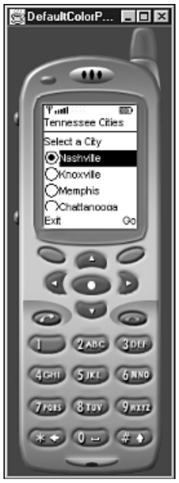


FIGURE 8.12
The ChoiceGroup component.

To create a gauge, use the `Gauge()` constructor, which looks like this:

```
Gauge(String label, boolean interactive, int maxValue, int initialValue)
```

The first parameter is the standard label for the gauge, which should come as no surprise to you at this point. The `interactive` parameter is a Boolean value that determines whether the gauge is interactive. The `maxValue` parameter establishes the maximum range for the gauge, and finally, the `initialValue` parameter sets the gauge's initial value. Following is an example of how you might create a gauge to display a percentage of a lengthy task:

```
form = new Form("Working...");
Gauge statusGauge = new Gauge("Status", false, 100, 40);
form.append(statusGauge);
```

The results of this code are shown in Figure 8.13, where the gauge is at 40% of its maximum value.

For a gauge to prove useful, you must be able to alter its current value. You access the current value of a gauge using the `getValue()` and `setValue()` methods, whose prototypes follow:

```
int getValue()
void setValue(int value)
```

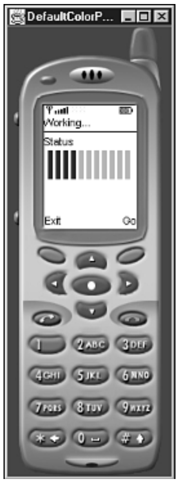


FIGURE 8.13
The Gauge component is used to create a visual bar graph that represents the status of a task.

It is also possible to manipulate the maximum value for a gauge by calling the `getMaxValue()` and `setMaxValue()` methods. The prototypes for these methods follow:

```
int getMaxValue()
void setMaxValue(int value)
```

Using Commands

Before seeing a practical example of how to use MIDP components to build a complete graphical user interface, there is one last topic to address. Commands provide the user with a means of invoking actions that are particular to the functionality of a specific MIDlet. Commands enable the user to do things such as navigate through a user interface, save data, and exit MIDlets. Just about anything a MIDlet might enable you to do can be made accessible using a command.

NEW TERM *command*—A GUI construct used to provide the user with a means of invoking an action that is particular to the functionality of a specific MIDlet.

Commands are modeled by the `Command` class, which provides a simple constructor that you use to create commands:

```
Command(String label, int commandType, int priority)
```

The `Command()` constructor accepts a label parameter that enables you to assign a label to the command. The label is important because it is what the user actually sees whenever they want to invoke the command. The `commandType` parameter specifies the type of the command, and must be one of several built-in command type constants, which you learn about in a moment. The last parameter to the constructor is `priority`, which specifies the priority level of the command. The priority level of a command is important because it determines how commands are displayed to the user. We will see more on command priority in just a moment.

The type of a command must be set to one of the following built-in command constants, which are defined in the `Command` class:

- **`OK`**—Verifies that it is okay to proceed with an action.
- **`CANCEL`**—Cancels an action.
- **`STOP`**—Stops an action.
- **`EXIT`**—Exits a MIDlet.
- **`BACK`**—Returns the user to the previous screen.
- **`HELP`**—Requests online help.
- **`ITEM`**—An application-specific command that is relative to a specific item on the current screen.
- **`SCREEN`**—An application-specific command that is relative to the current screen.

Command types are needed because devices can provide special buttons for certain functions such as the Back button. If a special button is made available for the Back operation, a command of type `BACK` will automatically be associated with this button.

The priority of a command is important because it is used to determine the placement of the command for user access. To understand why this is necessary, consider the fact that most mobile devices have limited buttons available for application-specific use. Consequently, only the most important commands are mapped to these soft buttons. If a MIDlet has other commands of lesser priority they will still be available, but only from menus within the MIDlet that aren't as easily accessible as clicking a soft button. The priority value of a command decreases with the level of importance; for example a value of 1 is assigned to commands with the highest priority. The specific placement of commands according to their priority is handled entirely by the application manager. This means you don't have any direct control over how commands will be presented to the user; your best bet is to use priority levels wisely. Keep in mind that priority values are entirely relative, so if you have a MIDlet with only one command, its priority level doesn't really matter.

To create a command, use the `Command()` constructor and pass along the three parameters you just learned about. Following is an example of how to create a Back command that would be used to move back to a previous screen in a MIDlet:

```
Command backCommand = new Command("Back", Command.BACK, 2);
```

Notice that the code in this example uses the BACK constant to establish the type of the command. After you've created a command, you then add it to a screen using the addCommand() method, which accepts the Command object as its only argument. Following is an example of how you would add the Back command to a screen using the addCommand() method:

```
screen.addCommand(backCommand);
```

The command is now part of the screen and will be displayed so that the user can access it. However, you haven't written any code to respond to the command. This is accomplished by setting a command listener for the screen containing the command, which is typically the MIDlet class itself. In this case the MIDlet class must implement the CommandListener interface, which contains a single method—commandAction(). Following is an example of how to set the command listener for a screen to the current MIDlet class:

```
screen.setCommandListener(this);
```

If the MIDlet class is to implement the CommandListener interface, it must say so in its declaration, which must look something like this:

```
public class MyMIDlet extends MIDlet implements CommandListener {
```

Of course, the real work in implementing the CommandListener interface is defining the commandAction() method, which has the following prototype:

```
void commandAction(Command c, Displayable s)
```

As you can see, the commandAction() method provides you with a Command object, along with a Displayable object that contains the screen for the command. Because only one commandAction() method is in a MIDlet, it is necessary to check for each command in the method before doing anything else. Following is an example of how you would handle both the Exit and Back commands in the commandAction() method:

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == backCommand) {
    // Go back!
  }
}
```

In this example I deliberately left the Back command handler empty because it will be unique for every MIDlet. However, notice how the two commands are both handled within the same commandAction() method.

The Mortgage Sample MIDlet

Finally, at long last, you can take a little bit of a breather and see some of your newfound knowledge put to use! This section works through the design and development of a MIDlet that calculates a monthly mortgage payment. My mom is a real estate agent and regularly uses a mobile phone to coordinate meetings and show houses. She also regularly must provide home buyers with an estimate regarding how much their monthly payment will be based upon how much they intend to borrow, the current interest rate, and the length of the loan. A MIDlet to perform this calculation on my mom's mobile phone would be incredibly handy to both her and her clients. So, you can think of this MIDlet as a little gift to my mom and all other real estate agents who wish they could calculate amortization tables in their heads.

The Mortgage MIDlet is a good example of how to use GUI components because it requires both the input and output of information. It must first input information about the mortgage using text fields on a form. From there, the mortgage calculations are carried out and the results displayed using an alert. One of the trickiest aspects of the Mortgage MIDlet involves a problem you likely wouldn't have thought about going into the development of the MIDlet. Floating point math isn't supported in MIDP programming, so the calculation of the monthly mortgage payment must be carefully carried out with integers. This is somewhat of a feat because all of the traditional equations used for this calculation involve floating point math.

The User Interface

The user interface for the Mortgage MIDlet primarily consists of a form that inputs the loan amount, interest rate, loan term, and monthly taxes and insurance from the user. These four pieces of information are retrieved through text fields, which are stored as member variables in the `Mortgage` class. Following are all of the member variables declared in the `Mortgage` class:

```
private Command exitCommand, goCommand;
private Display display;
private Form screen;
private TextField amountField, rateField, termField, tniField;
```

The commands are first declared in the MIDlet class. As you can see, two commands are in the MIDlet: `Exit` and `Go`. The `Exit` command exits the MIDlet, while the `Go` command tells the MIDlet to perform the mortgage calculation, which is then displayed using an alert. The `Display` object for the MIDlet is declared next, along with the `Form` object for the main screen. From there, the member variables are concluded with four `TextField` objects used to retrieve the mortgage information from the user.

With the member variables in place, you can move along to the `Mortgage()` constructor, which sets up the user interface for the MIDlet, as shown in Listing 8.1.

LISTING 8.1 The `Mortgage()` Constructor Initializes the Mortgage MIDlet

```
public Mortgage() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the Exit and Go commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  goCommand = new Command("Go", Command.OK, 2);

  // Create the screen form
  screen = new Form("Enter Loan Info");
  amountField = new TextField("Loan Amount ($)", "100000", 7,
➥TextField.NUMERIC);
  screen.append(amountField);
  rateField = new TextField("Interest Rate (%)", "8.5", 5, TextField.ANY);
  screen.append(rateField);
  termField = new TextField("Loan Term (years)", "15", 2, TextField.NUMERIC);
  screen.append(termField);
  tniField = new TextField("Taxes & Ins. ($)", "1200", 5, TextField.NUMERIC);
  screen.append(tniField);

  // Set the Exit and Go commands for the screen
  screen.addCommand(exitCommand);
  screen.addCommand(goCommand);
  screen.setCommandListener(this);
}
```

The `Mortgage()` constructor starts off by retrieving the `Display` object for the MIDlet, which is necessary later for setting the current display in the `startApp()` method. The `Exit` and `Go` commands are then created with an equal priority level of 2. The bulk of the constructor is then spent creating the text fields for the loan information. You'll notice that the types of the fields are set to accommodate the numeric information, except for the interest rate field, which must allow decimal places. Also, the text fields include default values so that you can quickly try out a mortgage calculation when you first run the MIDlet. The last step in the `Mortgage()` constructor is to set the `Exit` and `Go` commands for the screen, as well as set the MIDlet class as a command listener for the commands. With the commands created and the MIDlet class set as the command listener, you're ready to see how the commands are handled by the MIDlet.

Handling Commands

The Exit and Go commands in the Mortgage MIDlet are handled in the commandAction() method. The Exit command needs to exit the MIDlet, while the Go command must retrieve the loan information from the text fields, parse them into numeric values, calculate the monthly mortgage payment, and then display the result. Although this sounds like a lot of work, the commandAction() method delegates some of the messier tasks to helper methods. Listing 8.2 shows the code for the commandAction() method:

LISTING 8.2 The commandAction() Method Handles Commands for the Mortgage MIDlet

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == goCommand) {
    // Calculate the monthly payment
    int amount = Integer.parseInt(amountField.getString());
    int rate1K = parseRate(rateField.getString());
    int term = Integer.parseInt(termField.getString());
    int tni = Integer.parseInt(tniField.getString());
    int payment = calcPayment(amount, rate1K, term, tni);

    // Display the monthly payment
    Alert paymentAlert = new Alert("Mortgage", "Monthly Payment: $" +
      String.valueOf(payment), null, AlertType.INFO);
    paymentAlert.setTimeout(Alert.FOREVER);
    display.setCurrent(paymentAlert);
  }
}
```

You've already seen the Exit command handler code several times, so I won't bore you with repeat coverage. Instead, let's jump straight into the Go command handler and see what's going on. Because the loan information is stored in the text fields as text, it is necessary to parse them out as numbers to use them in the mortgage calculation. One problem with this approach is that the user will likely enter an interest rate as a floating-point number with decimal places. There is no way to work with floating-point numbers in a MIDlet, so the solution is to doctor the number by shifting the decimal to the right and working with the number as if it were in the thousands. In other words, an interest rate of 7.25% gets converted to 7,250. As long as the mortgage calculation takes this into account, you've successfully dodged the floating-point problem.

The helper method that shifts the decimal on the interest rate is called `parseRate()`, and is shown in Listing 8.3.

LISTING 8.3 The `parseRate()` Method Parses an Integer Interest Rate from a String

```
private int parseRate(String strRate) {
  // Convert the string to a string buffer
  StringBuffer str = new StringBuffer(strRate);

  // See if there is a decimal point in the rate, and delete it
  int i;
  for (i = 0; i < str.length(); i++)
    if (str.charAt(i) == '.') {
      str.deleteCharAt(i);
      break;
    }

  // Pad the rate string so that it is exactly 4 digits
  for (i = str.length(); i < 4; i++)
    str.append("0");

  return Integer.parseInt(str.toString());
}
```

The main emphasis of the Mortgage MIDlet is its user interface, so I'll leave it to you to study the `parseRate()` method on your own if you want to know the details of how the number is being parsed. Remember that the role of the method is to take a string containing a number with up to three decimal places and return a four-digit integer representation of the number.

Note

The `parseRate()` method is smart enough to enable the user to enter an interest rate without a decimal because the decimal character is hard to get to on some devices. This means that an interest rate entered as "85" will be interpreted as 8.5%, and returned by the `parseRate()` method as the integer 8,500.

Returning to the `commandAction()` method, the mortgage calculation is carried out by the `calcPayment()` method, which you learn about in a moment. The resulting monthly mortgage payment is then formatted and displayed in an alert with an infinite timeout, which means the user must voluntarily exit the alert to return to the loan information form.

Crunching the Numbers

The mortgage calculation is carried out in the `calcPayment()` method, which is surprisingly lean considering how tricky it was to determine how to carry out a mortgage calculation with integer math. Listing 8.4 shows the magical code that performs the mortgage calculation without the luxury of floating-point numbers.

LISTING 8.4 The `calcPayment()` Method Calculates the Monthly Mortgage Payment Without Using Floating-Point Math

```
private int calcPayment(int amount, int rate1K, int term, int tni) {
  int monthlyPayment;
  long precision = 1000000l;
  int termMonths = term * 12;
  long tmp1 = (precision * 1200000l) / (1200000l + rate1K);
  long tmp2 = tmp1;

  // Calculate the principal and interest
  for (int i = 0; i < termMonths; i++)
    tmp2 = (tmp1 * tmp2) / precision;
  monthlyPayment = (int)(((precision * amount * rate1K) / 1200000l) /
    (precision - tmp2));

  // Add in the monthly taxes and insurance
  monthlyPayment += tni / 12;

  return monthlyPayment;
}
```

The basic premise behind the `calcPayment()` method is that any number with decimal places must be multiplied by a factor of 10 to shift the decimal right until the number is an integer. The trick with this approach comes into play because the mortgage equation involves raising a number to a fairly high power, which quickly overruns the Java number system when you deal with the kinds of large numbers that result from shifting the decimal place to the right. The solution is to carefully shift the decimal place back and forth throughout the equation so that you can work with integers but not blow out the Java number system with huge numbers. I'd love to continue on for pages with a detailed explanation of how the original mortgage equation translated into this code, but I won't. Suffice it to say that the code gets the job done!

Putting It All Together

You've now seen the pieces and parts of code that compose the Mortgage MIDlet. However, it might help to see all of the code in perspective so that you can fully understand how it fits together. Listing 8.5 contains the complete code for the Mortgage MIDlet.

LISTING 8.5 The Complete Source Code for the Mortgage MIDlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
import javax.microedition.io.*;

public class Mortgage extends MIDlet implements CommandListener {
  private Command exitCommand, goCommand, backCommand;
  private Display display;
  private Form screen;
  private TextField amountField, rateField, termField, tniField;

  public Mortgage() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit and Go commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    goCommand = new Command("Go", Command.OK, 2);

    // Create the screen form
    screen = new Form("Enter Loan Info");
    amountField = new TextField("Loan Amount ($)", "100000", 7,
➥TextField.NUMERIC);
    screen.append(amountField);
    rateField = new TextField("Interest Rate (%)", "8.5", 5, TextField.ANY);
    screen.append(rateField);
    termField = new TextField("Loan Term (years)", "15", 2, TextField.NUMERIC);
    screen.append(termField);
    tniField = new TextField("Taxes & Ins. ($)", "1200", 5, TextField.NUMERIC);
    screen.append(tniField);

    // Set the Exit and Go commands for the screen
    screen.addCommand(exitCommand);
    screen.addCommand(goCommand);
    screen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the screen
    display.setCurrent(screen);
  }
```

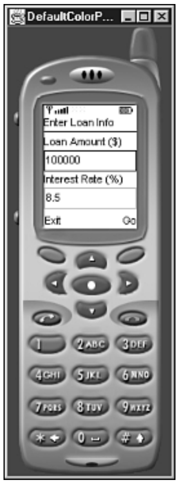
LISTING 8.5 continued

```
  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == goCommand) {
      // Calculate the monthly payment
      int amount = Integer.parseInt(amountField.getString());
      int rate1K = parseRate(rateField.getString());
      int term = Integer.parseInt(termField.getString());
      int tni = Integer.parseInt(tniField.getString());
      int payment = calcPayment(amount, rate1K, term, tni);

      // Display the monthly payment
      Alert paymentAlert = new Alert("Mortgage", "Monthly Payment: $" +
        String.valueOf(payment), null, AlertType.INFO);
      paymentAlert.setTimeout(Alert.FOREVER);
      display.setCurrent(paymentAlert);
    }
  }

  private int parseRate(String strRate) {
    // Convert the string to a string buffer
    StringBuffer str = new StringBuffer(strRate);

    // See if there is a decimal point in the rate, and delete it
    int i;
    for (i = 0; i < str.length(); i++)
      if (str.charAt(i) == '.') {
        str.deleteCharAt(i);
        break;
      }

    // Pad the rate string so that it is exactly 4 digits
    for (i = str.length(); i < 4; i++)
      str.append("0");

    return Integer.parseInt(str.toString());
  }

  private int calcPayment(int amount, int rate1K, int term, int tni) {
    int monthlyPayment;
    long precision = 1000000l;
```

LISTING 8.5 continued

```
      int termMonths = term * 12;
      long tmp1 = (precision * 1200000l) / (1200000l + rate1K);
      long tmp2 = tmp1;

      // Calculate the principal and interest
      for (int i = 0; i < termMonths; i++)
        tmp2 = (tmp1 * tmp2) / precision;
      monthlyPayment = (int)(((precision * amount * rate1K) / 1200000l) /
        (precision - tmp2));

      // Add in the monthly taxes and insurance
      monthlyPayment += tni / 12;

      return monthlyPayment;
    }
  }
```

You have already seen the vast majority of this code, but here you see it as it appears in the source code file and how the complete MIDlet came together. You are now ready to see the MIDlet in action. After compiling, pre-verifying, and packaging the Mortgage MIDlet, you can take it for a test spin in the J2ME emulator. Figure 8.14 shows the Mortgage MIDlet running in the emulator.



Enter Loan Info
Loan Amount ($)
100000
Interest Rate (%)
8.5
Exit Go

FIGURE 8.14
The Mortgage MIDlet prompts you to enter the loan information.

The Mortgage MIDlet begins by prompting you to enter the loan information. Of course, the default values are already set, so you can quickly test the MIDlet by selecting the `Go` command and viewing the results. Figure 8.15 shows the resulting monthly mortgage payment calculated by the Mortgage MIDlet. It makes you want to pay cash for your next house, huh?

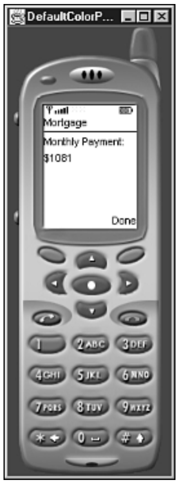


FIGURE 8.15
The Mortgage MIDlet calculates and displays the monthly mortgage payment based on the loan information entered.

That wraps up the Mortgage MIDlet, which I hope you found useful as a practical example of using MIDP GUI components. If you have any friends in the real estate business, feel free to pass along the MIDlet to them and maybe they'll give you a break on the commission when you make your next big purchase with them!

Summary

This lengthy lesson covered a great deal of territory. You learned about MIDlet GUIs and how they are constructed. In fact, you learned about virtually every GUI class in the MIDP API, which means you now have at least a basic understanding of all aspects of MIDlet GUIs.

This lesson introduced you to the display. From there, you learned about screens and forms and how they serve as the basis for most MIDlet GUIs. You then were introduced to all of the MIDlet GUI components, which offer a wide range of features. You learned about commands and how actions within a MIDlet are accessed and handled. Finally, the lesson guided you through the design and development of a handy MIDlet that calculates mortgage payments.

The next lesson shifts your attention toward networking, and introduces you to the MIDlet classes and interfaces that facilitate I/O and networking.

Q&A

Q Is it possible to use any of the standard Java AWT components in MIDlets?

A No. The GUI classes covered throughout this lesson are pretty much the extent of the MIDP GUI features available for constructing MIDlets. The idea behind the MIDP GUI classes is to keep things simple and compact, which is extremely important given the display and resource constraints imposed on mobile devices.

Q Is it okay to mix screen-based components with form-based components?

A Absolutely. Screen-based components such as `Alert`, `List`, and `TextBox` must be used the same way you would use a form, meaning that they take up the entire display area. Even so, there is no reason you can't switch back and forth between these components and custom forms that contain form-based components (items) such as string items, text fields, and choice groups. The key to successfully merging all of these components is to think of your MIDlet's user interface in terms of discrete screens. The Mortgage MIDlet is actually a good example of how these components can be mixed—the input screen is a custom form, whereas the output screen is an alert.

Q Do any of the MIDP GUI components generate standard Java events?

A No. The delegation event model employed by J2SE and J2EE has been more or less replaced in MIDP programming by commands. Although commands don't offer nearly the flexibility of standard Java events, they are very simple and they get the job done.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. With respect to GUIs, what class does every MIDlet have a unique instance of?
2. What are the only two direct subclasses of the `Displayable` class?
3. How do you know if a GUI component can be placed on a form?
4. What is the primary difference between the `List` class and the `ChoiceGroup` class?

Exercises

1. Create a new payment screen for the Mortgage MIDlet that is a form containing a single string item.
2. Modify the Mortgage MIDlet so that it displays the monthly mortgage payment in the new string item, and supports a `Back` command for moving from the payment screen back to the loan information screen.

DAY 9

Exploring MIDlet I/O and Networking

Perhaps the most intriguing aspect of MIDlet design is the capability of accessing the Internet with a wireless connection. Although wireless networking might seem like a difficult task, in reality there is little difference at the programming level between wired and wireless networking. In fact, the MIDP classes and interfaces used for wireless networking are very similar to those used in standard J2SE and J2EE networking applications. Granted, the architecture of the MIDP classes is a little different, as you learned in Day 5, "Working within the CLDC and MIDP APIs," but the general use of the classes is similar to traditional Java network programming. Even if you haven't done any network development in standard Java, I think you'll find MIDlet networking pretty easy to grasp.

This lesson leads you through the fundamentals of MIDlet networking, highlighting the most important facets of the MIDP API that are used to create networked MIDlets. Toward the end of the lesson, you will create a complete sample MIDlet that obtains data over a network connection and displays it to the user. The following major topics are covered in this lesson:

- Becoming acquainted with the basics of MIDP networking
- Analyzing the classes associated with MIDlet I/O
- Learning how to perform MIDP network communications
- Designing and constructing an interesting networking MIDlet

Inside MIDP Networking

As you learned in Day 5, the Connected Limited Device Configuration (CLDC) makes somewhat of a departure from the networking approach in the standard J2SE API by defining a Generic Connection Framework (GCF). The purpose of the GCF is to provide a level of abstraction for networking services, which helps in enabling different devices to only support network protocols specific to their needs. The GCF ultimately acts as an optimization for mobile devices that simply do not have enough resources to support all of the network protocols that are part of J2SE networking. Additionally, some high-end J2SE networking features such as multicast sockets are not necessary on mobile devices. Instead of forcing all devices to support all network protocols, the GCF describes a framework of network support from which a device can pick and choose what protocols and services it wants to support.

In reality, it is not up to the device itself to select what networking features it will support; instead, it is up to the device profile. The role of the GCF is to describe the various networking features available to all devices, along with an extensible system in which different device profiles can support a subset of these features. Although it is structured somewhat differently, the GCF is implemented as a functional subset of the J2SE API. The GCF describes one fundamental class named `Connector` that is used to establish all network connections. Specific types of network connections are modeled by GCF interfaces that are obtained through the `Connector` class. The `Connector` class and the connection interfaces are located in the `javax.microedition.io` package. Descriptions of these interfaces follow and will serve as a recap of what you learned in Day 5:

- **`Connection`**—A basic connection that can only be opened and closed
- **`ContentConnection`**—A stream connection that provides access to Web data
- **`DatagramConnection`**—A datagram connection suitable for handling packet-oriented communication
- **`InputConnection`**—An input connection to a communications device
- **`OutputConnection`**—An output connection to a communications device

- **`StreamConnection`**—A two-way connection to a communications device
- **`StreamConnectionNotifier`**—A special notification connection that is used to wait for a stream connection to be established

As you might notice, these interfaces are still somewhat general and are not too close to approximating a real network protocol. They do offer a basic architecture that is capable of supporting a wide range of networking features. Even though different network protocols require different approaches at the coding level, the CLDC interfaces provide a uniform approach to carrying out network operations. One advantage of the GFC is that application-specific network code adheres to a similar structure regardless of the network protocol employed.

The limitation of the GFC is that it only goes so far in describing network features. The specifics of these features are left up to device profiles. For example, the MIDP API builds on these interfaces to support networking in MIDlets. More specifically, the MIDP API adds the `HttpConnection` interface, which adds a functional component onto the GCF that supports HTTP network connections. HTTP connections enable MIDlets to connect to Web pages. The MIDP specification dictates that HTTP connections are the only types of connections absolutely required of a MIDP implementation. In other words, you can only count on having HTTP networking capabilities from within a MIDlet. This is somewhat of a limitation, and definitely impacts the design of MIDlets.

Note

Sun's MIDP implementation included in the J2ME Wireless Toolkit follows the MIDP specification by only supporting the HTTP protocol for networking. However, Motorola goes a step further by supporting additional (optional) networking protocols in their Motorola SDK for J2ME. This means that you have more flexibility in developing networking MIDlets using the Motorola toolkit. Of course, the networking support on a specific physical device ultimately determines whether your MIDlet will run properly. If you plan to specifically target Motorola devices, you'll be able to rely on the additional networking features. However, if you want to target other devices, you might want to stick with the strict MIDP specification and only utilize HTTP connections.

You always use the `Connector` class to establish network connections, regardless of the connection type. All the methods in the `Connector` class are static, with the most important one being the `open()` method. The three versions of the `open()` method are

```
static Connection open(String name) throws IOException
static Connection open(String name, int mode) throws IOException
static Connection open(String name, int mode, boolean timeouts)
  throws IOException
```

The first parameter to these methods is the connection string. It is the most important parameter because it determines the type of connection being made. The connection string describes the connection by adhering to the following general form:

`Scheme``:`*`Target`*`[;`*`Parameters`*`]`

The *`Scheme`* parameter is the name of the network protocol, such as `http` or `ftp`. The *`Target`* parameter is typically the name of the network address for the connection, but can vary according to the specific protocol. The last parameter, *`Parameters`*, is a list of parameters associated with the connection. Some examples of different types of connection strings for various network connections are

- **HTTP**—`"http://www.incthegame.com/"`
- **Socket**—`"socket://www.incthegame:1800"`
- **Datagram**—`"datagram://:9000"`
- **File**—`"file:/Stats.txt"`
- **Port**—`"comm:0;baudrate=9600"`

Note

The syntax of the connection string accepted by the `open()` method must adhere to the standard Uniform Resource Indicator (URI) syntax, which is described on the Web at `http://www.ietf.org/rfc/rfc2396.txt`. I have to warn you that this is a pretty technical document, so I only recommend viewing it if you absolutely must know the intricate details of how connection strings are formatted.

Keep in mind that although these examples are accurate in terms of describing possible connection strings, the only one of them that has guaranteed support in a given MIDP implementation is the first one. This is because the MIDP specification only requires an implementation to support HTTP connections. If you happen to know that a given MIDP implementation supports a particular type of connection, you can certainly take advantage of it. Otherwise, you will need to stick with HTTP connections.

The second and third versions of the `open()` method support a second parameter, `mode`, which describes the mode of the connection. The mode of a network connection describes whether the connection is being opened for reading, writing, or both. The following constants are defined in the Connector class, and can be used to describe the mode of the connection: `READ`, `WRITE`, or `READ_WRITE`. If you use the first version of the `open()` method that doesn't take the `mode` parameter, the `READ_WRITE` mode is used by default.

The last version of the `open()` method accepts a third parameter, `timeouts`, which is a flag that indicates whether the code that called the method is capable of handling a timeout exception. This exception is thrown if a timeout period for establishing the connection fails. The details of the timeout period and how it is handled are left up to the specific protocol, so you don't have much control over how it is carried out.

The `open()` method returns an object of type `Connection`, which is the base interface for all of the other connection interfaces. To use a certain kind of connection interface, you cast the `Connection` interface returns by `open()` to the appropriate type. The following line of code illustrates how to open an HTTP connection using the `StreamConnection` interface:

```
StreamConnection conn = (StreamConnection)Connector.open(
  "http://www.incthegame.com/Fortunes.txt");
```

Later in the lesson, you will learn how to use this stream connection to obtain an input stream and read data.

Understanding MIDlet I/O

The `Connector` class and its associated connection interfaces are used to obtain a connection to a network through a particular protocol. After the connection is established, you must use an I/O class to actually read to or write from the connection. These I/O classes are supported in the `java.io` package:

- **`ByteArrayInputStream`**—An input stream that buffers its input in an internal byte array
- **`ByteArrayOutputStream`**—An output stream that buffers its output in an internal byte array
- **`DataInputStream`**—A stream from which data can be read into primitive Java data types
- **`DataOutputStream`**—A stream to which data can be written from primitive Java data types
- **`InputStream`**—The base class for all input streams
- **`InputStreamReader`**—A stream from which data can be read as characters of text
- **`OutputStream`**—The base class for all output streams
- **`OutputStreamWriter`**—A stream to which data can be written as characters of text

- **`PrintStream`**—An output stream that facilitates the output of individual primitive data types
- **`Reader`**—An abstract class for reading character streams
- **`Writer`**—An abstract class for writing character streams

If you have any familiarity with standard J2SE I/O, these classes probably look pretty familiar. Perhaps the two most important classes in this list are `InputStream` and `OutputStream`, which provide the basic functionality for inputting and outputting data from and to streams. The next two sections explain how to use these classes, which provide the functional basis for the other I/O classes.

The `InputStream` Class

The `InputStream` class is an abstract class that serves as the base class for the other MIDP input stream classes. `InputStream` defines a basic interface for reading streamed bytes of information. The typical scenario when using an input stream is to create an `InputStream` object from a connection and then tell it you want to input information by calling an appropriate `read()` method. If no input information is currently available, the input stream uses a technique known as *blocking* to wait until input data becomes available. An example of blocking is the case of using an input stream to read information from an HTTP connection. Until the Web server delivers the information, no input is available to the `InputStream` object. The `InputStream` object then waits (blocks) until information becomes available, at which point the information is processed as input data.

NEW TERM *blocking*—An I/O technique that involves an input or output stream waiting until data has been received or sent before allowing an application to continue.

The `InputStream` class defines the following methods:

- `read()`
- `read(byte b[])`
- `read(byte b[], int off, int len)`
- `skip(long n)`
- `available()`
- `mark(int readlimit)`
- `reset()`
- `markSupported()`
- `close()`

As you can see, the `InputStream` class defines three different methods for reading input data in various ways. The first `read()` method takes no parameters and simply reads a byte of data from the input stream and returns it as an integer. This version of `read()` returns -1 if the end of the input stream is reached. Because this version of read returns a byte of input as an `int`, you must cast it to a `char` if you are reading characters. The second version of `read()` takes an array of bytes as its only parameter, enabling you to read multiple bytes of data at once. The data that is read is stored in this array. This version of `read()` returns the actual number of bytes read or -1 if the end of the stream is reached. The last version of `read()` takes a byte array, an integer offset, and an integer length as parameters. This version of `read()` is similar to the second version except that it enables you to specify where in the byte array you want to place the information that is read. The `off` parameter specifies the offset into the byte array to start placing read data, and the `len` parameter specifies the maximum number of bytes to read.

The `skip()` method is used to skip over bytes of data in the input stream. `skip()` takes a `long` value n as its only parameter, which specifies how many bytes of input to skip. It returns the actual number of bytes skipped or -1 if the end of the input stream is reached.

The `available()` method is used to determine the number of bytes of input data that can be read without blocking. `available()` takes no parameters and returns the number of available bytes. This method is useful if you want to ensure that there is input data available before calling the `read()` method, and therefore avoid blocking.

The `mark()` method marks the current position in the stream. You can later return to this position using the `reset()` method. The `mark()` and `reset()` methods are useful in situations in which you want to read ahead in the stream but not lose your original position. Notice that the `mark()` method takes an integer parameter, `readlimit`. `readlimit` specifies how many bytes can be read before the mark becomes invalidated. In effect, `readlimit` determines how far you can read ahead and still be able to reset the marked position. The `markSupported()` method returns a Boolean value representing whether an input stream supports the mark/reset functionality.

Finally, the `close()` method closes an input stream and releases any resources associated with the stream. It is not necessary to explicitly call `close()` because input streams are automatically closed when the `InputStream` object is destroyed.

The `OutputStream` Class

The `OutputStream` class is the output counterpart to `InputStream`, and serves as an abstract base class for all the other MIDP output stream classes. `OutputStream` defines the basic protocol for writing streamed data to an output device. You typically create an

OutputStream object on a connection and then call the write() method to tell it you want to output information. The OutputStream class uses a blocking technique similar to the one used by InputStream: It blocks until data has been written to an output device. While blocking (waiting for the current output to be processed), the OutputStream class does not allow any further data to be output.

The OutputStream class supports the following methods:

- write(int b)
- write(byte b[])
- write(byte b[], int off, int len)
- flush()
- close()

Similar to the read() methods in the InputStream class, the OutputStream class defines three different write() methods for writing data in several different ways. The first write() method writes a single byte to the output stream, as specified by the integer parameter b. The second version of write() takes an array of bytes as a parameter and writes it to the output stream. The last version of write() takes a byte array, an integer offset, and a length as parameters. This version of write() is very much like the second version except that it uses the other parameters to determine where in the byte array to begin outputting data, along with how much data to output. The off parameter specifies an offset into the byte array from which you want to begin outputting data, and the len parameter specifies how many bytes are to be output.

The flush() method is used to flush the output stream; calling flush() forces the OutputStream object to output any pending data.

The close() method closes an output stream and releases any resources associated with the stream. As with InputStream objects, it isn't usually necessary to explicitly call close() on an OutputStream object because streams are automatically closed when they are destroyed.

Communicating Across Network Connections

You now understand both the connection aspect of MIDP networking, along with the I/O classes that are necessary to facilitate reading from or writing to a network connection. All that's missing is putting this knowledge together and actually communicating across a network connection. Fortunately, this task is probably much easier than you might have imagined. Communicating over a network connection in a MIDlet basically consists of the following three steps:

1. Establish a network connection using the `Connector` class.
2. Obtain a stream on the connection using one of the connection interfaces.
3. Read data from or write data to the stream using one of the I/O stream classes.

That's really all there is to it. Of course, the real work comes into play when you attempt to do something meaningful with the data sent or received over the connection. This part of the networking equation is entirely MIDlet-specific, so I can't really provide any general guidelines. However, I can explain how to perform the three basic steps of MIDlet network communication, which are covered in the next few sections.

Making the Connection

The first step in performing any kind of network communication in a MIDlet involves establishing a network connection. As you learned earlier in the lesson, this is accomplished through a call to the static `open()` method on the `Connector` class. The easiest way to open a connection through the `open()` method is to use the version of the method that accepts a single string parameter consisting of a connection string. The following example demonstrates how to open an HTTP stream connection on a Web page:

```
StreamConnection conn = null;
conn = (StreamConnection)Connector.open("http://www.incthegame.com/news.htm");
```

This code opens an HTTP connection on the Web page located at `http://www.incthegame.com/news.htm`. The connection is cast to a `StreamConnection` object so that you can communicate over the connection by an input or output stream. The next step is to obtain an input or output stream on the connection, which is necessary to actually perform any I/O.

Obtaining an Input Stream

Obtaining a stream on a connection is very straightforward. For the purposes of this example, I will demonstrate how to obtain an input stream, which can be used to read the Web page associated with the HTTP connection. The code necessary to obtain an input stream on the connection, through which you can read data, is as follows:

```
InputStream in = null;
in = conn.openInputStream();
```

The `openInputStream()` method is all that is necessary to establish an input stream on a stream connection. After you have obtained an `InputStream` object, you can begin reading data from the HTTP connection.

Reading from the Stream

You learned a little earlier that the `read()` method is the key to reading data from an `InputStream` object. The basic version of the `read()` method reads a byte of data from the input stream and returns it as an integer. Subsequent calls to the `read()` method result in additional bytes of data being read from the stream, continuing until the end of the stream is reached, which results in the `read()` method returning –1. This means that you can create a loop that continually calls the `read()` method until a return value of –1 is encountered. The following is an example of how to code such a loop that reads lines of data from an input stream and prints them to standard output:

```
StringBuffer data = new StringBuffer();
int ch;
while ((ch = in.read()) != -1) {
  if (ch != '\n') {
    // Read the line a character at a time
    data.append((char)ch);
  }
  else {
    // Display the line of text data
    System.out.println(data.toString());

    // Clear the string for the next line
    data = new StringBuffer();
  }
}
```

In this code the `data` variable is a string buffer that is used to hold a line of data at a time. The `read()` method is called to read characters from the stream until the end of the stream is reached. Within the `while` loop, the current character is compared to `'/n'` (newline) to see whether the end of the line has been reached. If so, you know that you have accumulated a complete line of text, so it is okay to print it to standard output and clear the string buffer.

This code represents the basic premise of how you might read data through an input stream on an HTTP connection. If you consider how much data is floating around out there in Web pages, you can see that there are many possibilities to be explored when it comes to mining the Web for data with MIDlets. In fact, the next two lessons (Day 10, "Being Your Own Wireless Meteorologist" and Day 11, "Wireless Navigation") give you excellent examples of how to build practical MIDlets that extract data from live Web sites.

The Fortune Example MIDlet

Admittedly, this lesson has focused somewhat on networking theory and hasn't given you a whole lot of practical coverage of performing I/O over a connection in a MIDlet. Granted, you just learned how to read from an HTTP connection and display the results, but you haven't seen MIDP networking performed within the context of a functioning

MIDlet. We remedy this problem by working through the design and development of a MIDlet named Fortune that reads and displays random quotes (*fortunes*) from a file located on a remote Web site. The Fortune MIDlet will use an HTTP connection to open a text file and read its contents into a string vector. An individual fortune is then selected from the vector and displayed on the main screen of the MIDlet.

NEW TERM *fortune*—A quote or witticism, often humorous, that is intended to provoke thought.

Note Strictly speaking, you might think of a fortune as more of a prediction, as in the fortunes that come with fortune cookies. However, the concept of a fortune software program dates back to the early days of Unix when fortune programs would display interesting quotes.

The idea behind the Fortune MIDlet is to establish a network connection and read lines of text from a text file located on the World Wide Web. The text file itself must be located on a Web server somewhere on the Web and be readily accessible. In other words, you should be able to open the file in a Web browser. For the purposes of testing the MIDlet, I placed the file `Fortunes.txt` on my own Web site at `http://www.incthegame.com`. You could place your version of this file on your own Web site. The contents of the `Fortunes.txt` file are as follows:

```
We learn from history that we do not learn from history.
Few men have virtue to withstand the highest bidder.
Petty laws breed great crimes.
The right to be let alone is the beginning of all freedom.
America's one of the finest countries anyone ever stole.
What's the Constitution between friends?
The highest result of education is tolerance.
Tyranny is always better organized than freedom.
```

Note Keep in mind that you are free to create your own text file with all your own witty sayings and euphemisms. Incidentally, the quotes in my `Fortunes.txt` file aren't my own, but I found them all interesting enough to share.

The key to the fortunes text file is that each fortune must appear on a line by itself. This is necessary because the Fortune MIDlet processes each line of text and assumes that each fortune is on a line of its own. This makes the parsing of the text much easier, and our goal is to always take the easy way out whenever possible! Although there are no set rules on how to develop a MIDlet such as Fortune, the following is one possible breakdown of the MIDlet's functional components:

- The user interface
- Handling commands
- Retrieving Fortune data

The next few sections take you through the development of the code for each of these aspects of the Fortune MIDlet, along with a wrap-up where you put everything together.

The User Interface

When the Fortune MIDlet is first run, it retrieves all the fortunes from the fortunes text file on the network. It then selects one of the fortunes at random and displays it to the user. If you want to view a different fortune, select the `Next` command, which selects another fortune at random and displays it in place of the previous fortune. The only other command the Fortune MIDlet needs is the `Exit` command, which allows you to exit the MIDlet. These commands are created as member variables in the `Fortune` class along with the main screen for the MIDlet, the string item that holds the current fortune, and the vector of fortune strings:

```
private Command exitCommand, nextCommand;
private Display display;
private Form screen;
private StringItem fortuneItem;
private Vector fortunes;
```

The first variables declared are the command variables, which represent the `Exit` and `Next` commands. The `Display` object for the MIDlet is then declared, which is necessary to set the MIDlet's screen. The `screen` variable is then declared as a `Form` object, which is required so that it can contain a string item in which to display a fortune. The only other GUI component for the MIDlet is `fortuneItem`, which is a string item that actually displays the current fortune string. The last member variable for the MIDlet is `fortunes`, a collection class of type `java.lang.Vector`. If you aren't familiar with the `Vector` class, it basically provides a means of storing multiple objects; it is like an array that is capable of growing as you add elements items to it.

After declaring the member variables, you can move on to the `Fortune()` constructor, whose code is shown in Listing 9.1.

LISTING 9.1 The `Fortune()` Constructor Initializes the Fortune MIDlet

```
public Fortune() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the Exit and Next commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  nextCommand = new Command("Next", Command.OK, 2);

  // Create the main screen form
  screen = new Form("Fortune of the Day");
  fortuneItem = new StringItem("", "Reading fortunes...");
  screen.append(fortuneItem);

  // Set the Exit and Next commands for the screen
  screen.addCommand(exitCommand);
  screen.addCommand(nextCommand);
  screen.setCommandListener(this);

  // Create the fortunes vector
  fortunes = new Vector();
}
```

The constructor first obtains a `Display` object for the MIDlet, and then creates the two MIDlet commands (`Exit` and `Next`). The main screen is then created as a `Form` object, along with the `StringItem` component used to display the current fortune. The string item is added to the screen through a call to the `append()` method. The `Exit` and `Go` commands are then associated with the screen by adding the `exitCommand` and `goCommand` objects using the `addCommand()` method. The last step of the constructor is to create a `Vector` object for storing the fortune strings.

The `Fortune()` constructor does most of the work of establishing the MIDlet's user interface. However, some work is still left for the `startApp()` method to take care of—initialization of the fortune strings and display of the first fortune. Listing 9.2 contains the code for the `startApp()` method, which handles these chores with two simple method calls.

LISTING 9.2 The `startApp()` Method Loads the Fortunes and Displays the First Random Fortune

```
public void startApp() throws MIDletStateChangeException {
  // Set the current display to the location screen
  display.setCurrent(screen);
```

LISTING 9.2 continued

```
    // Initialize the fortunes vector
    readFortunes();

    // Show the first random fortune
    showFortune();
  }
```

The `startApp()` method first sets the current display, which is fairly standard code for most MIDlets. It then calls the `readFortune()` method to retrieve the fortunes from the network, followed by `showFortune()` to display the first fortune. You will learn how these methods work in a few moments. For now, let's take a look at how the MIDlet's commands are handled.

Handling Commands

The Fortune MIDlet includes two commands: `Exit` and `Next`. Both of these commands are handled in the `commandAction()` method, whose code is shown in Listing 9.3.

LISTING 9.3 The `commandAction()` Method Handles Commands for the Fortune MIDlet

```
  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == nextCommand) {
      // Show the next random fortune
      showFortune();
    }
  }
```

By now that you should be comfortable with the role of the `Exit` command, as well as the code that makes it function properly. We focus instead on the `Next` command, which notifies the MIDlet to display another random fortune. As you can see in the code, there isn't a whole lot to focus on—the `commandAction()` method calls the `showFortune()` method to show the next random fortune. The next section explains exactly how the `showFortune()` method works, as well as the `readFortunes()` method, which is in fact the more significant of the two.

Retrieving and Showing the Fortunes

In case you have forgotten, the Fortune MIDlet is intended to demonstrate the networking capabilities of MIDlets. All of the networking functionality of the Fortune MIDlet is wrapped up in the `readFortunes()` method, which reads all the fortunes in the `Fortunes.txt` file and stores them in a fortunes vector. To make the code a little more understandable, consider the major tasks required of the `readFortunes()` method:

1. Open a network connection to the `Fortunes.txt` file.
2. Read data from the network connection a character at a time until a complete line of text is read.
3. For each line of text in the file, create a new string and add it to the vector.

The second and third tasks in this list are performed within a `while` loop that continues until the end of the file is reached. Listing 9.4 contains the code for the `readFortunes()` method, which reveals how these tasks are carried out.

LISTING 9.4 The `readFortunes()` Method Reads the Fortunes from a Text File

```
private void readFortunes() {
  StreamConnection conn = null;
  InputStream in = null;
  StringBuffer data = new StringBuffer();

  try {
    // Open the HTTP connection
    conn = (StreamConnection)Connector.open("http://www.incthegame.com/
➥Fortunes.txt");

    // Obtain an input stream for the connection
    in = conn.openInputStream();

    // Read a line at a time from the input stream
    int ch;
    boolean done = false;
    while ((ch = in.read()) != -1) {
      if (ch != '\n') {
        // Read the line a character at a time
        data.append((char)ch);
      }
      else {
        // Add the fortune to the fortunes vector
        fortunes.addElement(data.toString());

        // Clear the string for the next line
        data = new StringBuffer();
      }
    }
  }
  catch (IOException e) {
    System.err.println("The connection could not be established.");
  }
}
```

The readFortunes() method starts off by creating some member variables that are used throughout the method. The conn variable is a StreamConnection object that stores the stream connection for the fortune text file. The in variable is an InputStream object that is used to read from the text file. The last variable, data, is a string buffer that is used to store an individual line of the fortune file as it is being read.

Note

Be sure to change the URL for the Fortunes.txt file to a location of your own where you can place the file for testing. If you have access to a Web server, copy the file to the server and take note of its URL. Then make sure the URL in the code for the readFortunes() method matches the URL for the file and you should be ready to go.

The readFortunes() method begins by opening a stream connection on the URL of the Fortunes.txt file by calling the static open() method of the Connector class. The stream returned by the open() method is cast to a StreamConnection object. The openInputStream() method is then called on the stream connection to open the input stream from which data can be read. At this point an input stream is opened on the fortunes text file and you are ready to start reading data.

The while loop in the readFortunes() method carries out the details of reading each fortune from the text file as individual lines of text. A line of text is assembled in the string buffer by reading one character after another until the end of the line is reached. After the full line is read, the addElement() method is called on the fortunes vector to add the fortune string. The string buffer is then cleared so that the next line of the file (the next fortune) can be read. The remainder of the fortunes text file is read in this same manner, which systematically builds a vector containing all of the fortune strings. As this code hopefully demonstrates, reading data across the Web through the MIDP networking API isn't too difficult.

In addition to the readFortunes() method, the Fortune MIDlet also calls the showFortune() method to show a random fortune. Listing 9.5 shows the code for this method.

LISTING 9.5 The showFortune() Method Selects and Displays a Random Fortune

```
private void showFortune() {
  // Check to make sure the fortunes vector isn't empty
  if (!fortunes.isEmpty()) {
    // Create and seed the random number generator
    Random rand = new Random(Calendar.getInstance().getTime().getTime());
```

LISTING 9.5 continued

```
      // Set a random fortune
      int fortuneNum = Math.abs(rand.nextInt()) % fortunes.size();
      fortuneItem.setText((String)fortunes.elementAt(fortuneNum));
    }
    else
      fortuneItem.setText("No fortune!");
  }
```

The `showFortune()` method first checks to see whether the fortunes vector is empty, in which case it displays the text `"No fortune!"`. This is essentially an error condition because it means that the MIDlet was unable to read any fortunes. In the event that the fortunes vector does contain fortune strings, the `Random` and `Calendar` classes are used to obtain a random positive integer. The purpose of the `Calendar` class is to retrieve the current time, which serves as a unique seed for the `Random` object. The random number obtained through the `Random` object is then used to determine a random fortune number, `fortuneNum`. This number serves as an index into the fortune vector, `fortunes`. The selected fortune is used as the parameter to the `setText()` method of the `fortuneItem` string item, which results in the fortune being displayed on the screen.

Note

The `Random` class implements a random number generator that is useful for obtaining random numbers. The key to getting random numbers with the `Random` class is to seed the random number generator with a different number each time a MIDlet is run. One tricky way to do this is to use the current time as the seed for the random number generator. Because the current time is specified down to the second, you are virtually always guaranteed to get a unique integer number.

Putting It All Together

Thus far you've seen how all the parts of the Fortune MIDlet work on their own. Like Dr. Frankenstein, you've collected all the body parts and are ready to put the entire monster together. The complete code for the Fortune MIDlet is contained in Listing 9.6; it shows how the different methods you've learned about fit together:

LISTING 9.6 The Complete Source Code for the Fortune MIDlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
```

Listing 9.6 continued

```
public class Fortune extends MIDlet implements CommandListener {
  private Command exitCommand, nextCommand;
  private Display display;
  private Form screen;
  private StringItem fortuneItem;
  private Vector fortunes;

  public Fortune() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit and Next commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    nextCommand = new Command("Next", Command.OK, 2);

    // Create the main screen form
    screen = new Form("Fortune of the Day");
    fortuneItem = new StringItem("", "Reading fortunes...");
    screen.append(fortuneItem);

    // Set the Exit and Next commands for the screen
    screen.addCommand(exitCommand);
    screen.addCommand(nextCommand);
    screen.setCommandListener(this);

    // Create the fortunes vector
    fortunes = new Vector();
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the location screen
    display.setCurrent(screen);

    // Initialize the fortunes vector
    readFortunes();

    // Show the first random fortune
    showFortune();
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
```

LISTING 9.6 continued

```
      notifyDestroyed();
    }
    else if (c == nextCommand) {
      // Show the next random fortune
      showFortune();
    }
  }

  private void readFortunes() {
    StreamConnection conn = null;
    InputStream in = null;
    StringBuffer data = new StringBuffer();

    try {
      // Open the HTTP connection
      conn = (StreamConnection)Connector.open("http://www.incthegame.com/
➥Fortunes.txt");

      // Obtain an input stream for the connection
      in = conn.openInputStream();

      // Read a line at a time from the input stream
      int ch;
      boolean done = false;
      while ((ch = in.read()) != -1) {
        if (ch != '\n') {
          // Read the line a character at a time
          data.append((char)ch);
        }
        else {
          // Add the fortune to the fortunes vector
          fortunes.addElement(data.toString());

          // Clear the string for the next line
          data = new StringBuffer();
        }
      }
    }
    catch (IOException e) {
      System.err.println("The connection could not be established.");
    }
  }

  private void showFortune() {
    // Check to make sure the fortunes vector isn't empty
    if (!fortunes.isEmpty()) {
      // Create and seed the random number generator
      Random rand = new Random(Calendar.getInstance().getTime().getTime());
```

LISTING 9.6 continued

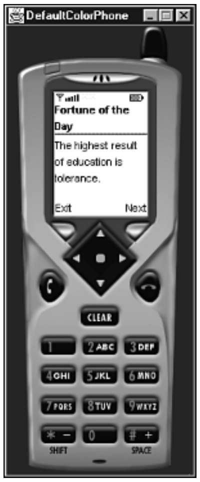
```
        // Set a random fortune
        int fortuneNum = Math.abs(rand.nextInt()) % fortunes.size();
        fortuneItem.setText((String)fortunes.elementAt(fortuneNum));
      }
      else
        fortuneItem.setText("No fortune!");
    }
  }
```

This is without a doubt the longest code listing you've encountered thus far in the book, so please don't be intimidated. If you study the code, you'll notice that virtually all of it is familiar because you just worked through the creation of most of it. Although the Fortune MIDlet certainly isn't the most complex networking MIDlet you'll ever encounter, it is a good starting point for learning how to utilize HTTP connections to read and process Web data.

Testing the Fortune MIDlet

Just as with any MIDlet, you must compile, preverify, and package the Fortune MIDlet before testing it. After you've performed these steps, I encourage you to try out the MIDlet using the J2ME emulator. Keep in mind that you must have the `Fortunes.txt` file available on a Web server somewhere, and the URL for it must be specified accurately in the `readFortunes()` method. Figure 9.1 shows the Fortune MIDlet after it first starts in the J2ME emulator.

FIGURE 9.1



Fortune of the Day

The highest result of education is tolerance.

The Fortune MIDlet displays a random fortune upon startup.

To view another random fortune, select the `Next` command. Figure 9.2 shows the Fortune MIDlet as it displays another fortune.

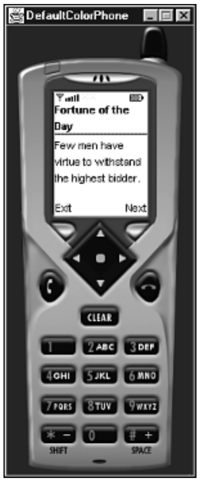


FIGURE 9.2
Upon selecting the Next command, the Fortune MIDlet displays another random fortune.

When you are tired of reading the fortunes, you can exit the MIDlet by selecting the `Exit` command. Congratulations, your first wireless networking MIDlet is now completed and ready to roll!

Summary

In this lesson you were introduced to networking as it applies to MIDlets. You began the lesson by learning about the Generic Connection Framework (GFC) that forms the basis of the CLDC network architecture. Built on top of this framework are the specific networking features made possible by the MIDP API. You learned in this lesson how these features impact networked MIDlets, as well as the MIDP I/O classes. After you gained a grasp on the theory and API aspects of MIDP networking, you moved on to learning how to establish a network connection, obtain an input stream, and read data from an HTTP connection. From there, you used what you learned to create a fully functional MIDlet that reads fortunes across a network from a text file and displays them to the user at random.

Although you covered a lot of territory, this lesson really only serves as an appetizer to the real networking entrées that are delivered in the next two lessons. More specifically, the next lesson guides you through the design and development of a MIDlet that is capable of retrieving and displaying live weather conditions for any U.S. city.

Q&A

Q How do I perform types of networking other than HTTP, such as datagram networking?

A To guarantee that your MIDlets will run on any MIDP device, you can't perform any networking beyond HTTP networking because there is no guarantee that the devices will support it. This isn't to say that some devices will support other types of networking, such as datagram networking. In fact, several Motorola phones already have support for datagrams. To utilize other types of networking in your MIDlets, you must target those specific devices and not worry about whether the MIDlets will run on other devices that don't support the network features. This might not work out for you depending on the type of application you're developing and the target user.

Q The Fortune MIDlet used the HTTP protocol to read a raw text file from a Web server and process its contents. Is it also possible to read a Web page that is stored in HTML form, and process its contents?

A Absolutely. In fact, that is the true intent of the HTTP protocol that is supported by all MIDP implementations. In reality, the Fortune MIDlet is bending the rules a little by reading a raw text file through HTTP. The next two lessons demonstrate how to read HTML data and parse information from it.

Q This lesson only demonstrated how to read data over a network connection through HTTP. Is it also possible to write data using a similar approach?

A Yes. Just as the `InputStream` class was used to read data through an HTTP connection, the `OutputStream` class can also be used to write data through an HTTP connection.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What fundamental GFC networking class is used to establish all MIDlet network connections?
2. What is the only type of network connection guaranteed to be supported by all MIDP implementations?

3. To establish an HTTP connection, what must you pass into the `Connector.open()` method?
4. How do you obtain an input stream on a connection after the connection is established?

Exercises

1. Try adding additional fortunes to the `Fortunes.txt` file and taking note of the changes when you run the Fortune MIDlet. Make sure you transfer the modified fortunes file to a Web server so that it is accessible from the Fortune MIDlet.
2. Modify the Fortune MIDlet so that it displays the source of the fortune underneath the fortune on the screen. This involves adding another string item to the screen so that it holds the source of the current fortune. Of course, this also means adding another vector that holds the source string of each respective fortune, not to mention changing the format of the `Fortunes.txt` file. The best way to change this file is to interpret each pair of lines as a fortune followed by its source.

DAY 10

Being Your Own Wireless Meteorologist

One of the most interesting applications of J2ME is retrieving live data from the Internet and making it readily available to the user. One such type of live data is the current weather conditions for a given city. Let's pretend that you're trying to decide whether to check your umbrella with your luggage as you prepare to board a flight. A quick check of the weather conditions in the city of your destination will give you an idea as to whether you need to have the umbrella handy. This information is available readily at your fingertips through J2ME and your wireless mobile device.

This lesson is one of several application-oriented lessons that focus on specific MIDlet applications. In this lesson, you design and build a weather MIDlet that is capable of retrieving live weather conditions for a city and state that you enter. Following are the major topics addressed while designing and building this MIDlet:

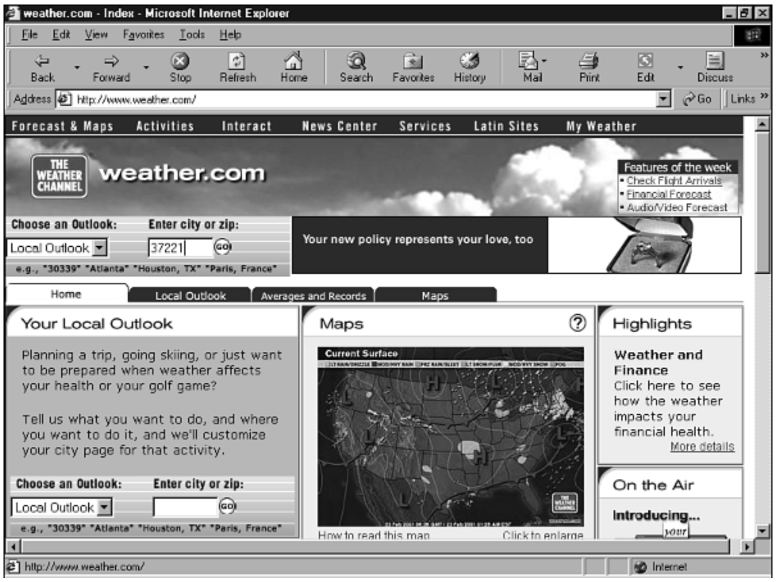
- Accessing weather information on the Web
- Analyzing the inner workings of a weather Web page
- Designing and coding the Weather MIDlet
- Testing the Weather MIDlet

Weather and the Web

As you might know, weather conditions are nothing new to the Web. A variety of different Web sites include up-to-the-minute weather information, readily accessible from a Web browser. Most of these sites require you to enter a city and state or a ZIP code. Figure 10.1 shows the Weather.com Web site, which prompts you for such information to retrieve local weather conditions.

FIGURE 10.1

The Weather.com Web site requires you to enter a city and state or ZIP code to view local weather conditions.

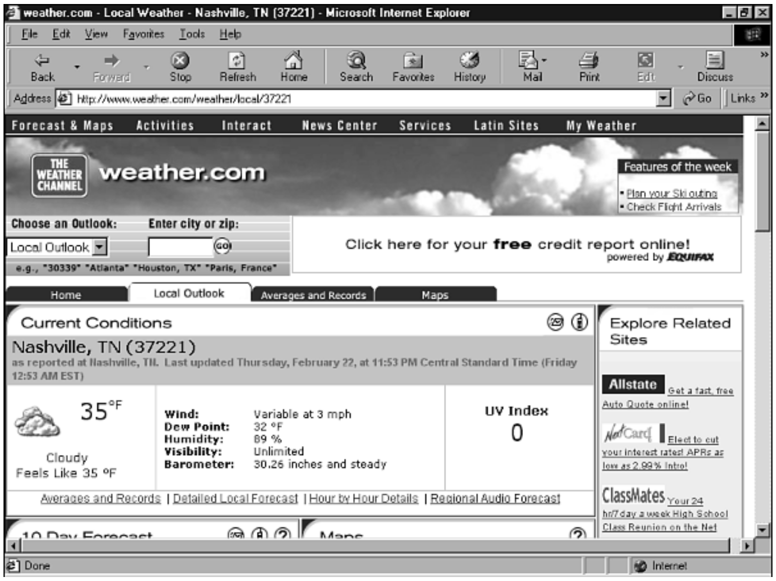

weather.com - Index - Microsoft Internet Explorer
http://www.weather.com/
Forecast & Maps Activities Interact News Center Services Latin Sites My Weather
weather.com
Choose an Outlook: Local Outlook
Enter city or zip: 37221
Your Local Outlook
Maps
Highlights

If you enter a city and state or ZIP code on the Weather.com Web site, the local weather conditions will appear in an easy-to-read format, as shown in Figure 10.2.

Although the weather conditions shown on the Weather.com Web site are appealing to the eye, it's hard to imagine them fitting within the constraints of a mobile device display. Additionally, you need a full-blown Web browser to view the Web page, which isn't entirely an option on many mobile devices. Granted, it's possible to use WAP to view a modified Web page; in fact, this is already being done. However, I look for more versatility in being able to retrieve any piece of weather information and display it within a MIDlet.

FIGURE 10.2

The Weather.com Web site displays local weather conditions in an easy-to-read format.


weather.com - Local Weather - Nashville, TN (37221) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.weather.com/weather/local/37221
Go Links
Forecast & Maps Activities Interact News Center Services Latin Sites My Weather
THE WEATHER CHANNEL weather.com
Features of the week
• Plan your Ski outing
• Check Flight Arrivals
Choose an Outlook: Enter city or zip:
Local Outlook
e.g., "30339" "Atlanta" "Houston, TX" "Paris, France"
Click here for your free credit report online! powered by EQUIFAX
Home Local Outlook Averages and Records Maps
Explore Related Sites
Allstate Get a fast, free Auto Quote online!
NextCard Elect to cut your interest rates! APRs as low as 2.99% Intro!
ClassMates Your 24 hr/7 day a week High School Class Reunion on the Net
Averages and Records | Detailed Local Forecast | Hour by Hour Details | Regional Audio Forecast
10 Day Forecast
Maps
Done
Internet

Current Conditions

Nashville, TN (37221)

as reported at Nashville, TN. Last updated Thursday, February 22, at 11:53 PM Central Standard Time (Friday 12:53 AM EST)

35°F
Cloudy
Feels Like 35 °F

Wind:	Variable at 3 mph
Dew Point:	32 °F
Humidity:	89 %
Visibility:	Unlimited
Barometer:	30.26 inches and steady

UV Index
0

Note

The Weather.com Web site is an excellent source of weather information, but its information is presented in too complex a manner to easily extract the weather data within a MIDlet. The idea is to find the information in a format that can be easily processed.

To display live weather conditions in a MIDlet, it is necessary to find a weather source with data that is in a format suitable for reading and displaying. Such a Web site is the National Weather Service's Internet Data Source Web site, also known as Weather.gov. The Weather.gov Web site is accessible through the URL `http://weather.gov/`, as shown in Figure 10.3.

The unique thing about the Weather.gov Web site is that it provides weather data in a simplified format that is easy to parse and display in your own applications. You can read a page of data from this site and extract pertinent pieces of weather information without too much trouble. This is the approach you will use in this lesson to display live weather conditions within a MIDlet. To understand this, let's look at a sample page on the Weather.gov site that shows current conditions for a city and state. Figure 10.4 shows the current (as of this writing!) weather conditions of my home state, Tennessee.

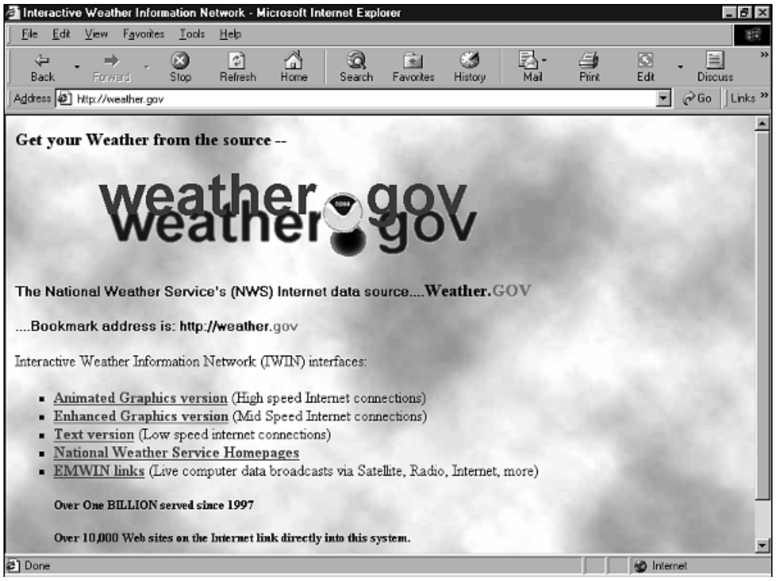


FIGURE 10.3

The National Weather Service's Weather.gov Web site provides live national weather conditions.

Interactive Weather Information Network - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://weather.gov
Go Links
weather.gov
Done
Internet

Get your Weather from the source --

The National Weather Service's (NWS) Internet data source....Weather.GOV

....Bookmark address is: http://weather.gov

Interactive Weather Information Network (IWIN) interfaces:

- Animated Graphics version (High speed Internet connections)
- Enhanced Graphics version (Mid Speed Internet connections)
- Text version (Low speed internet connections)
- National Weather Service Homepages
- EMWIN links (Live computer data broadcasts via Satellite, Radio, Internet, more)

Over One BILLION served since 1997

Over 10,000 Web sites on the Internet link directly into this system.

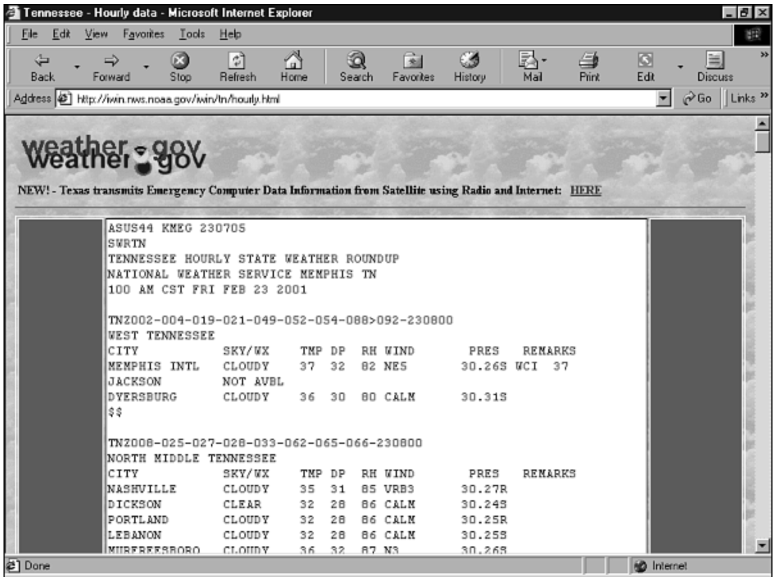


FIGURE 10.4

The National Weather Service's Weather.gov Web site includes a Web page for each state that lists live weather conditions for each major city.

Tennessee - Hourly data - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://iwin.nws.noaa.gov/iwin/tn/hourly.html
Go Links
weather.gov
Done
Internet

NEW! - Texas transmits Emergency Computer Data Information from Satellite using Radio and Internet: HERE

```
ASUS44 KMEG 230705
SWRTN
TENNESSEE HOURLY STATE WEATHER ROUNDUP
NATIONAL WEATHER SERVICE MEMPHIS TN
100 AM CST FRI FEB 23 2001

TNZ002-004-019-021-049-052-054-088>092-230800
WEST TENNESSEE
CITY            SKY/WX     TMP DP  RH WIND       PRES   REMARKS
MEMPHIS INTL    CLOUDY     37  32  82 NE5       30.26S WCI  37
JACKSON         NOT AVBL
DYERSBURG       CLOUDY     36  30  80 CALM      30.31S
$$

TNZ008-025-027-028-033-062-065-066-230800
NORTH MIDDLE TENNESSEE
CITY            SKY/WX     TMP DP  RH WIND       PRES   REMARKS
NASHVILLE       CLOUDY     35  31  85 VRB3      30.27R
DICKSON         CLEAR      32  28  86 CALM      30.24S
PORTLAND        CLOUDY     32  28  86 CALM      30.25R
LEBANON         CLOUDY     32  28  86 CALM      30.25S
MURFREESBORO    CLOUDY     36  32  87 N3        30.26S
```

As you can see in Figure 10.4, a listing of cities and their local weather conditions is after the header information that is near the top of the page. The different pieces of weather data are conveniently organized into columns, which makes them relatively easy to parse out and display within a MIDlet. Following are the meanings of the pieces of weather information shown on the page, as indicated by the headers above each column:

- **`SKY/WX`**—General conditions (cloudy, rainy, and so forth)
- **`TMP`**—Temperature (in degrees Fahrenheit)
- **`DP`**—Dew point
- **`RH`**—Relative humidity
- **`WIND`**—Wind speed and direction (in miles per hour)
- **`PRES`**—Barometric pressure (in inches)
- **`REMARKS`**—Additional information

MIDlets have limited screen real estate to work with, which means that you don't necessarily need to include every possible piece of weather information. It is sufficient to include the general conditions, temperature, relative humidity, and wind speed and direction. If you're into the weather enough to be overly interested in the dew point and barometric pressure, you probably want to see a Doppler radar map—Get on the Web with a real Web browser if you need that much detail! Otherwise, the Weather MIDlet developed in this lesson provides a sufficient overview of the current weather conditions for a given city.

Referring to Figure 10.4, you might notice that the URL used to access the local weather conditions is not the same as the Weather.gov URL mentioned earlier. For some reason, when you access local weather conditions the base URL must be specified as `http://iwin.nws.noaa.gov/`. This URL ends up pointing to the same page as the `http://weather.gov/` URL. However, you must use the `http://iwin.nws.noaa.gov/` URL when directly accessing local weather conditions. You learn how this works in the next section.

Peeking Inside a Weather Web Page

Now that you have a live weather source with weather data organized in a format that is relatively easy to access, it's time to think about how to retrieve that information. First it's important to understand how the Weather.gov weather data is organized. Following is the URL that was used in the previous example to get the weather data for Tennessee:

```
http://iwin.nws.noaa.gov/iwin/tn/hourly.html
```

As you can see, this URL includes the base URL `http://iwin.nws.noaa.gov/`, followed by a couple of directories (`iwin/tn/`), and finally the Web page itself (`hourly.html`). The weather data is organized into directories according to each state, which is why this URL includes a directory named `tn`, the abbreviation for Tennessee.

The Web page itself is the same for all states; it is always hourly.html. To retrieve the weather conditions for a different state, all you must do is change the two-letter state abbreviation in the URL. For example, following is the URL to retrieve the current weather conditions in Arizona:

```
http://iwin.nws.noaa.gov/iwin/az/hourly.html
```

By assembling this URL as a string you could easily retrieve the Web page for any state. In fact, following is the code to put together such a URL string:

```
String url = "http://iwin.nws.noaa.gov/iwin/" + state + "/hourly.html";
```

In this code, the `state` variable is assumed to be another string that contains the two-letter state abbreviation in lowercase. By plugging in the `state` variable, the URL for the state's weather conditions is created. Of course, it is still necessary to open the URL through a network connection and retrieve the Web page data. You tackle this a little later in the lesson. But first, you need to understand a bit more about the format of this data.

Keep in mind that although the weather data is in a nice-looking format when the hourly.html Web page is viewed in a Web browser, it is still HTML code, not raw text. You might need to parse through HTML tags to get to the weather data for a given city. To better understand this challenge, take a look at the part of the HTML source code for the Tennessee weather data Web page:

LISTING 10.1 An Excerpt from the Weather Data Web Page

```
<HTML><HEAD><META HTTP-EQUIV="Refresh" CONTENT="900"><TITLE>Tennessee -
➥Hourly data</TITLE></HEAD><BODY BACKGROUND="/iwin/sky1bkg.gif">

<table border="0" width="100%">

<tr>

  <td width="100%"><img border="0" src="/iwin/smwthr.gif" width="200"
➥ height="48"></td>

</tr>

<tr>

  <td nowrap width="100%"><b><font size="2">

<H5>NEW! - Texas transmits Emergency Computer Data Information from Satellite
➥ using Radio and Internet: &nbsp; <A HREF="http://emwin.hcad.org">HERE</A>

&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;

</font></b></td></tr></table><HR>
```

LISTING 10.1 continued

```
<table border="1" width="100%" bordercolor="#FFFFFF">

<tr>

<td width="33%" bgcolor="#008080">&nbsp;</td>

 <td nowrap bgcolor="FFFFFF" bordercolor="#000000">

<TT><PRE>ASUS44 KMEG 232115
SWRTN
TENNESSEE HOURLY STATE WEATHER ROUNDUP
NATIONAL WEATHER SERVICE MEMPHIS TN
300 PM CST FRI FEB 23 2001

TNZ002-004-019-021-049-052-054-088>092-232200
WEST TENNESSEE
CITY           SKY/WX    TMP DP  RH WIND       PRES   REMARKS
MEMPHIS INTL   LGT RAIN  51  39  63 E10       30.18F
MILLINGTON     CLOUDY    48  38  68 E5        30.19F
JACKSON        CLOUDY    51  39  63 NE6       30.22F
DYERSBURG      FAIR      45  37  76 E9        30.25F
SOMERVILLE     FAIR      57  38  49 E3        30.19F
COVINGTON      CLOUDY    49  40  71 E10       30.20F
UNION CITY     SUNNY     49  39  68 E8        30.23F
BOLIVAR        MOSUNNY   52  37  56 NE7       30.19F
PARIS          SUNNY     48  37  65 E9        30.25F
HUNTINGDON     SUNNY     48  35  60 E8        30.24F
LEXINGTON      SUNNY     54  36  50 N7        30.22F
SELMER         FAIR      57  37  47 E9        30.20F
SAVANNAH       FAIR      61  33  34 E6        30.21F
$$

TNZ008-025-027-028-033-062-065-066-232200
NORTH MIDDLE TENNESSEE
CITY           SKY/WX    TMP DP  RH WIND       PRES   REMARKS
NASHVILLE      MOSUNNY   49  37  63 NE6       30.27F
DICKSON        SUNNY     50  32  50 E5        30.25S
PORTLAND       SUNNY     48  36  61 E5        30.27F
LEBANON        SUNNY     48  36  61 NE5       30.27F
SMYRNA         MOSUNNY   52  41  66 N3        30.28
MURFREESBORO   SUNNY     55  39  54 NE7       30.25
COOKEVILLE     SUNNY     59  37  44 N6        30.27F
LIVINGSTON     SUNNY     50  36  57 NE7       30.27F
GALLATIN       CLEAR     50  34  53 NE6       30.29F
CROSSVILLE     MOSUNNY   55  34  45 SE6       30.26S
$$

TNZ061-075-076-078-094-096-097-232200
SOUTH MIDDLE TENNESSEE
```

Listing 10.1 continued

```
CITY           SKY/WX    TMP DP  RH WIND       PRES   REMARKS
LEWISBURG      SUNNY     54  37  54 E6        30.25F
FAYETTEVILLE   SUNNY     59  37  44 CALM      30.23F
TULLAHOMA      SUNNY     59  37  44 CALM      30.23F
WINCHESTER     N/A       63  28  27 SE9       30.22F
SHELBYVILLE    SUNNY     55  39  54 NW3       30.25F
$$

TNZ017-018-042-046-047-232200
NORTHEAST TENNESSEE
CITY           SKY/WX    TMP DP  RH WIND       PRES   REMARKS
TRI CITIES     SUNNY     54  35  48 SW3       30.28F
MOUNTAIN CITY  SUNNY     54  27  35 W8        30.28R
$$

TNZ012-013-036-069-073-074-232200
EAST CENTRAL TENNESSEE
CITY           SKY/WX    TMP DP  RH WIND       PRES   REMARKS
KNOXVILLE      MOSUNNY   56  40  55 N9        30.26F
OAK RIDGE      MOSUNNY   55  37  50 CALM      30.30F
GATLINBURG     SUNNY     57  39  51 N7        30.29F
MORRISTOWN     SUNNY     57  37  47 CALM      30.30F
JACKSBORO      SUNNY     55  34  44 CALM      30.28F
$$

TNZ083-085-099-232200
SOUTHEAST TENNESSEE
CITY           SKY/WX    TMP DP  RH WIND       PRES   REMARKS
CHATTANOOGA    SUNNY     57  40  52 N8        30.25F
ATHENS         SUNNY     61  39  44 NE5       30.27F
$$

SKY/WX REPORTED AS FAIR MEANS FEW OR NO CLOUDS BELOW
12 THOUSAND FEET.
_
```

The beginning of the page includes much information not related to the specific weather conditions of each city in the state. However, down into the page a bit it is apparent that the weather data is organized to make it very easy to extract the data. More specifically, each line of data begins with the name of the city followed by each piece of weather data formatted into neat columns. This means that to extract the weather data for a given city, you can look at the beginning of each line of the HTML code and see whether it begins with the city name. If it does, you can jump across to each of the columns and extract each piece of weather information. This is the approach you use in the next section when you assemble the Weather MIDlet.

Note

A typical Weather.gov Web page for a state is considerably longer than the sample page shown here. However, much of the information is repeated throughout the page, which means that you can generally focus only on the first section of the page.

One interesting aspect of the weather HTML code. In the listing is a strange-looking square on the last line of code. This square represents an unprintable character (`'\003'`), which is a character that has no visual representation. The weather Web page uses this special character to denote different pages of data. For your purposes, it is sufficient to use this character as the end of the data because all of the cities appear on the first page of data.

Note

An unprintable character is not a character that is so profane that it can't be placed in print. An unprintable character literally has no visual representation, and therefore must be rendered as a generic shape, which in this case is a square. Unprintable characters often have a special meaning and are used to perform a certain control function. For example, the `'\n'` character technically has no visual representation, but it results in a carriage return, or newline, being placed in a document.

Constructing the Weather MIDlet

The premise behind the Weather MIDlet is to establish a network connection and retrieve a Web page containing weather data for a given city. The weather data Web page must then be parsed so that the specific pieces of weather information can be extracted and displayed to the user. In addition to being a good example of how to develop a practical-networked MIDlet, the Weather MIDlet also demonstrates how to retrieve information from the user and then display results using GUI components. In many ways, the Weather MIDlet is the most complete example you've tackled throughout the book thus far.

You can take many approaches to developing a MIDlet such as the Weather MIDlet, but I break the development down into three major parts:

- The user interface
- Handling commands
- Retrieving weather data

The next few sections dig into the details of each of these aspects of the Weather MIDlet, along with a wrap-up where you put it all together.

The User Interface

The Weather MIDlet is designed such that the user must be able to enter the city and state for which weather conditions are to be retrieved. After the information is obtained, a way is needed to present this information back to the user. This design dictates that there should be two screens in the MIDlet: one for handling the city and state user input and one for displaying the resulting weather conditions. The user enters the city and state in one screen and clicks a button, after which the weather conditions are retrieved and the results displayed in the other screen. The MIDlet is always displaying one of the two screens, which means the screens comprise the MIDlet's entire user interface.

The first screen allows the user to input the location (city and state) of the weather conditions, so I refer to it as the location screen. This screen needs to be a form containing two text fields that enable the user to enter the city and state. The second screen is more involved because it deals with displaying several pieces of weather information. Earlier in the lesson we decided to focus on the general conditions, temperature, relative humidity, and wind speed and direction. So the output screen must provide string items that can be used to display each of these pieces of information. We will refer to this screen as the conditions screen.

In addition to housing the input and output components for the MIDlet's user interface, the location and conditions screens must also support a few commands. More specifically, the location screen needs a `Go` command that initiates the retrieval of the weather data. Also, because the location screen serves more or less as the MIDlet's main screen, it is the screen that should have the `Exit` command that exits the MIDlet. Finally, the conditions screen needs a `Back` command that enables the user to return to the location screen.

Note

There is no reason you couldn't put an `Exit` command on the conditions screen, but it seems to make more sense in the Weather MIDlet to require the user to return to the locations screen with the `Back` command. This helps reinforce the notion that the locations screen is the main MIDlet screen.

The screen material is implemented using various GUI components that are stored as member variables of the Weather MIDlet class. The member variable declarations for this class are as follows:

```
private Command exitCommand, goCommand, backCommand;
private Display display;
private Form locationScreen;
private TextField cityField, stateField;
private Form conditionsScreen;
private StringItem locationItem, conditionsItem, temperatureItem,
  humidityItem, windItem;
```

The first variables declared are the command variables, which represent the `Exit`, `Go`, and `Back` commands, respectively. The `Display` object for the MIDlet is then declared, which is standard in all MIDlets. The `locationScreen` variable is then declared as a `Form` object, which makes sense given that it houses other GUI components. These other components consist of two `TextField` objects, `cityField` and `stateField`, which enable the user to input the city and state. The `conditionsScreen` variable is then declared as a form. And finally, several `StringItem` objects are created to display the various pieces of weather information.

With all the member variables declared for the MIDlet, we can initialize them in the MIDlet's constructor. Following is the code for the `Weather()` constructor.

LISTING 10.2 The `Weather()` Constructor

```
public Weather() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the Exit, Go, and Back commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  goCommand = new Command("Go", Command.OK, 2);
  backCommand = new Command("Back", Command.BACK, 2);

  // Create the location screen form
  locationScreen = new Form("Enter Location");
  cityField = new TextField("City", "", 25, TextField.ANY);
  locationScreen.append(cityField);
  stateField = new TextField("State", "", 2, TextField.ANY);
  locationScreen.append(stateField);

  // Set the Exit and Go commands for the location screen
  locationScreen.addCommand(exitCommand);
  locationScreen.addCommand(goCommand);
  locationScreen.setCommandListener(this);

  // Create the conditions screen form
  conditionsScreen = new Form("Current Conditions");
  locationItem = new StringItem("", "");
  conditionsScreen.append(locationItem);
  conditionsItem = new StringItem("", "");
```

LISTING 10.2 continued

```
    conditionsScreen.append(conditionsItem);
    temperatureItem = new StringItem("", "");
    conditionsScreen.append(temperatureItem);
    humidityItem = new StringItem("", "");
    conditionsScreen.append(humidityItem);
    windItem = new StringItem("", "");
    conditionsScreen.append(windItem);

    // Set the Back command for the conditions screen
    conditionsScreen.addCommand(backCommand);
    conditionsScreen.setCommandListener(this);
  }
```

After obtaining a `Display` object for the MIDlet, the constructor creates the three MIDlet commands (`Exit`, `Go`, and `Back`). The location screen is then created as a `Form` object, along with its `TextField` components. Each of these components is added to the screen with a call to the `append()` method. Because the `Exit` and `Go` commands are associated with the location screen, the `exitCommand` and `goCommand` objects are added to the screen as commands using the `addCommand()` method. We next look at the conditions screen.

The initialization of the conditions screen is very similar to that of the location screen, with the obvious exception that it consists of string items instead of text fields. The screen is first created as a `Form` object. Each string item is then created and added to the form. Notice that the string items are created with empty values, which is necessary because you will be building these strings from scratch when the weather data is retrieved. After creating and adding the string items to the screen, the `backCommand` object is added to the screen to establish the `Back` command.

The `Weather()` constructor handles all the work of setting up the user interface for the MIDlet. Even so, some work must still be done for the commands to work properly and enable you to navigate between the two screens.

Handling Commands

As you know, the Weather MIDlet has three screens: `Exit`, `Go`, and `Back`. You're already familiar with the `Exit` command from previous MIDlets, but the `Go` and `Back` commands are somewhat unique to the functionality of the Weather MIDlet. The `Go` command is defined for the location screen, and is responsible for initiating the gathering and displaying of the weather data after the user enters the city and state. The `Back` command, on the other hand, comes into play on the conditions screen, and returns the user back to the location screen to either enter another city and state or exit the MIDlet.

The method responsible for handling the commands and implementing their functionality is the `commandAction()` method, with the code shown in Listing 10.3.

LISTING 10.3 The `commandAction()` Method

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == goCommand) {
    // Get the conditions for the city and state
    getConditions(cityField.getString().toUpperCase(),
      stateField.getString().toLowerCase());
  }
  else if (c == backCommand) {
    // Clear the location fields
    cityField.setString("");
    stateField.setString("");

    // Set the current display back to the location screen
    display.setCurrent(locationScreen);
  }
}
```

The `Exit` command is first handled by comparing the command parameter with the `exitCommand` member variable. If there is a match, the MIDlet is exited. Of course, this is code that you've seen several times before. The code for the `Go` command is much more interesting in that it calls the `getConditions()` method to retrieve and display the current weather conditions for the specified city and state. Although you work through the details of the `getConditions()` method in the next section, it's worth taking a look at its prototype to understand how this code works:

```
void getConditions(String city, String state)
```

The `getConditions()` method takes the city and state as string parameters and uses them to get the weather data and display it on the conditions screen. If you recall from earlier in the lesson, the URL for the weather Web page uses the lowercase state abbreviation as one of the directory names; this explains why the value of the `stateField` string is converted to lowercase before being passed into the `getConditions()` method. The `cityField` string, on the other hand, is converted to uppercase so that it can be searched for within the weather data Web page.

The `Back` command code is pretty straightforward in that it first clears the city and state fields and then sets the current display to the location screen. This gives the user the opportunity to enter another city and state or exit the MIDlet.

Retrieving Weather Data

This brings us to the heart of the Weather MIDlet, the `getConditions()` method, which does the dirty work of opening the weather Web page, reading the data, parsing out the pertinent information, and displaying it on the conditions screen. In many ways, this method can serve as the basis for any MIDlet that must extract data from a Web page and display it to the user. Given the limited browser capabilities of mobile devices, this is a challenge you can find yourself facing again.

To better understand the `getConditions()` method, keep in mind that it must carry out the following tasks:

1. Construct a URL for the weather Web page.
2. Open a network connection to the URL.
3. Read data from the network connection and search for a matching city.
4. When a match is found, extract the pertinent weather information.
5. Use the weather information to set the conditions screen string items.
6. Set the current display to the conditions screen.

Although these steps aren't quite as easily distinguishable in the code for the `getConditions()` method, they are nonetheless carried out. Listing 10.4 contains the code for the `getConditions()` method.

LISTING 10.4 The `getConditions()` Method

```
private void getConditions(String city, String state) {
  StreamConnection conn = null;
  InputStream in = null;
  StringBuffer data = new StringBuffer();

  try {
    // Open the HTTP connection
    conn = (StreamConnection)Connector.open("http://iwin.nws.noaa.gov/iwin/" +
      state + "/hourly.html");

    // Obtain an input stream for the connection
    in = conn.openInputStream();

    // Read a line at a time from the input stream
    int ch;
    boolean done = false;
    while (((char)(ch = in.read()) != '\003') && (ch != -1) && !done) {
      if (ch != '\n') {
        // Read the line a character at a time
       data.append((char)ch);
```

LISTING 10.4 continued

```
        }
        else {
          // Make sure the line is long enough
          if (data.length() >= city.length()) {
            // See if the line starts with the city name
            if ((city.length() > 0) &&
              (data.toString().substring(0, city.length()).compareTo(city) == 0)) {
              // Fill in the conditions string items
              locationItem.setText(city + ", " + state.toUpperCase());
              conditionsItem.setText("Cond.: " +
                data.toString().substring(15, 22));
              temperatureItem.setText("Temperature: " +
                data.toString().substring(25, 27) + '\260');
              humidityItem.setText("Rel. Humidity: " +
                data.toString().substring(33, 35) + "%");
              windItem.setText("Wind: " +
                data.toString().substring(36, 44) + " mph");

              // Set the current display to the conditions screen
              display.setCurrent(conditionsScreen);

              // We're done, so bail out of the outer while loop
              done = true;
            }
          }
          // Clear the string for the next line
          data = new StringBuffer();
        }
      }

      // The done flag tells us if there was a problem
      if (!done)
        display.setCurrent(new Alert("Weather",
          "The location is invalid. Please try another.", null, AlertType.ERROR));
    }
    catch (IOException e) {
      System.err.println("The connection could not be established.");
    }
  }
```

The `getConditions()` method starts begins by creating a few member variables that are used throughout the method. The `conn` variable is a `StreamConnection` object that stores the stream connection used to read the weather Web page. The `in` variable is an `InputStream` object that is used to read from the Web page. And finally, the `data` variable is a string that is used to store one line of the Web page at a time as it is being read.

To read the Web page and parse out the weather information, the `getConditions()` method must first open a stream connection on the Web page's URL. The static `open()`

method of the `Connector` class establishes this connection, which is cast to a `StreamConnection` object. The `openInputStream()` method is then called on the stream connection to open the input stream from which data can be read. This brings us to the main section of the `getConditions()` method, which consists of a `while` loop that reads the Web page from an input stream.

The `while` loop in the `getConditions()` method basically reads a character at a time from the input stream, checking to make sure that it hasn't encountered the special `'\003'` character that ends the page, or that it hasn't run out of data. If it passes both of these tests, the character is checked to see whether it is a newline (`'\n'`). This is done to see whether the end of a line has been reached, in which case it is time to examine it. The examination of a line consists of checking to see whether it begins with the city name. If the city name isn't matched, the line of text is cleared so that a new line can be read. If the city name is matched, you know that the line contains the appropriate weather data and you can begin extracting it.

The weather data is aligned according to predefined columns, so it's very easy to extract it for display purposes. The `subString()` method of the `String` class is used to pull out the exact piece of weather data for each of the weather information string items (temperature, relative humidity, and so on). This information is then formatted a little and set to the string items so that it will be visible in the conditions screen. Of course, you must make the conditions screen visible to the user with a call to the `setCurrent()` method on the `Display` object. After that is accomplished, a special flag (`done`) is set so that the `while` loop exits and no more reading is performed on the input stream.

When the `while` loop exits, you can determine whether the weather information was successfully retrieved by checking the `done` flag. This check is performed so that an error message can be displayed; an `Alert` object is used to display the error message.

That wraps up the `getConditions()` method, which is the heart of the Weather MIDlet. Although the extraction of the weather data was a little tricky, this code is still surprisingly straightforward given the fact that the MIDlet is dynamically reading data from a live Web page.

Putting It All Together

You've now worked through the details of the Weather MIDlet and seen how its critical parts work. However, it is beneficial to see the complete code listing to get a feel for how everything fits together. Listing 10.5 represents the complete source code for the Weather MIDlet, which consists of the code you've already seen plus some familiar support code.

LISTING 10.5 The Weather MIDlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
import javax.microedition.io.*;

public class Weather extends MIDlet implements CommandListener {
  private Command exitCommand, goCommand, backCommand;
  private Display display;
  private Form locationScreen;
  private TextField cityField, stateField;
  private Form conditionsScreen;
  private StringItem locationItem, conditionsItem, temperatureItem,
    humidityItem, windItem;

  public Weather() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit, Go, and Back commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    goCommand = new Command("Go", Command.OK, 2);
    backCommand = new Command("Back", Command.BACK, 2);

    // Create the location screen form
    locationScreen = new Form("Enter Location");
    cityField = new TextField("City", "", 25, TextField.ANY);
    locationScreen.append(cityField);
    stateField = new TextField("State", "", 2, TextField.ANY);
    locationScreen.append(stateField);

    // Set the Exit and Go commands for the location screen
    locationScreen.addCommand(exitCommand);
    locationScreen.addCommand(goCommand);
    locationScreen.setCommandListener(this);

    // Create the conditions screen form
    conditionsScreen = new Form("Current Conditions");
    locationItem = new StringItem("", "");
    conditionsScreen.append(locationItem);
    conditionsItem = new StringItem("", "");
    conditionsScreen.append(conditionsItem);
    temperatureItem = new StringItem("", "");
    conditionsScreen.append(temperatureItem);
    humidityItem = new StringItem("", "");
    conditionsScreen.append(humidityItem);
    windItem = new StringItem("", "");
    conditionsScreen.append(windItem);
```

LISTING 10.5 continued

```
  // Set the Back command for the conditions screen
  conditionsScreen.addCommand(backCommand);
  conditionsScreen.setCommandListener(this);
}

public void startApp() throws MIDletStateChangeException {
  // Set the current display to the location screen
  display.setCurrent(locationScreen);
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == goCommand) {
    // Get the conditions for the city and state
    getConditions(cityField.getString().toUpperCase(),
      stateField.getString().toLowerCase());
  }
  else if (c == backCommand) {
    // Clear the location fields
    cityField.setString("");
    stateField.setString("");

    // Set the current display back to the location screen
    display.setCurrent(locationScreen);
  }
}

private void getConditions(String city, String state) {
  StreamConnection conn = null;
  InputStream in = null;
  StringBuffer data = new StringBuffer();

  try {
    // Open the HTTP connection
    conn = (StreamConnection)Connector.open("http://iwin.nws.noaa.gov/iwin/" +
      state + "/hourly.html");

    // Obtain an input stream for the connection
    in = conn.openInputStream();
```

LISTING 10.5 continued

```
      // Read a line at a time from the input stream
        int ch;
        boolean done = false;
        while (((char)(ch = in.read()) != '\003') && (ch != -1) && !done) {
          if (ch != '\n') {
            // Read the line a character at a time
            data.append((char)ch);
          }
          else {
            // Make sure the line is long enough
            if (data.length() >= city.length()) {
              // See if the line starts with the city name
              if ((city.length() > 0) &&
                (data.toString().substring(0, city.length()).compareTo(city) ==
                ➥0)) {
                // Fill in the conditions string items
                locationItem.setText(city + ", " + state.toUpperCase());
                conditionsItem.setText("Cond.: " +
                  data.toString().substring(15, 22));
                temperatureItem.setText("Temperature: " +
                  data.toString().substring(25, 27) + '\260');
                humidityItem.setText("Rel. Humidity: " +
                  data.toString().substring(33, 35) + "%");
                windItem.setText("Wind: " +
                  data.toString().substring(36, 44) + " mph");

                // Set the current display to the conditions screen
                display.setCurrent(conditionsScreen);

                // We're done, so bail out of the outer while loop
                done = true;
              }
            }
            // Clear the string for the next line
            data = new StringBuffer();
          }
        }

        // The done flag tells us if there was a problem
        if (!done)
          display.setCurrent(new Alert("Weather",
            "The location is invalid. Please try another.", null,
  AlertType.ERROR));
      }
      catch (IOException e) {
        System.err.println("The connection could not be established.");
      }
    }
  }
```

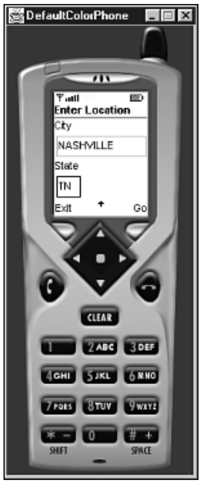
This is a long code listing; however, don't lose sight of the fact that this is a practical and powerful MIDlet in that it is establishing a connection to a Web server, reading data from a Web page, processing that data, and then displaying the results. And because of the inherent nature of J2ME and MIDP, all of this is taking place over a wireless connection on a mobile device. I would challenge you to try and accomplish this feat with another programming language in twice the amount of source code!

Testing the Weather MIDlet

After compiling, pre-verifying, and packaging the Weather MIDlet, you can begin checking weather conditions by running the MIDlet within the J2ME emulator. Figure 10.5 shows the Weather MIDlet running in the emulator after I've entered a city and state.

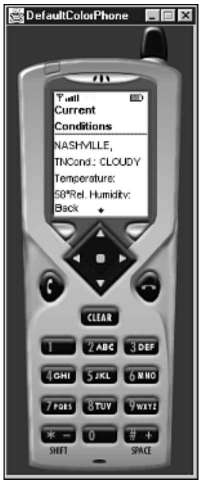
FIGURE 10.5

The Weather MIDlet prompts you to enter a city and state.



After entering a city and state, you can retrieve the current weather conditions by simply selecting the `Go` command. Figure 10.6 shows the result of selecting this command after entering Nashville, TN as the city and state.

Notice in Figure 10.6 that the current weather conditions are displayed in a clean format that is easy to read even on such a small display. To return to the location screen, select the `Back` command. You can then enter a new city and state and start all over, or exit the MIDlet.



```
Current
Conditions
NASHVILLE,
TNCond.: CLOUDY
Temperature:
58°Rel. Humidity:
Back
```

FIGURE 10.6
After entering a city and state, and selecting the Go *command, the Weather MIDlet displays current weather conditions.*

Summary

This lesson is the first of several that focus entirely on the design and development of a practical MIDlet. In this lesson you constructed a weather MIDlet that is capable of reading current weather conditions over a network connection and displaying them to the user. The Weather MIDlet makes use of many of the J2ME programming skills you've learned throughout the book thus far, including GUI and network programming. More specifically, the Weather MIDlet utilizes two GUI screens to input location information from the user and display the weather conditions. Also, your MIDP networking skills were put to use as the MIDlet made use of a network connection to read and process a Web page to extract weather data.

As the introduction to this lesson mentioned, the Weather MIDlet could prove useful any time you are traveling. In the next lesson you develop another practical travel MIDlet that makes a great companion to the Weather MIDlet—the Directions MIDlet, which enables you to enter two addresses and then uses the Web to retrieve the driving directions to travel from one place to the other. Using these two MIDlets together, not only will you not get lost the next time you hop in your car, but you'll know if you need to carry an umbrella!

Q&A

Q Could the weather information retrieved by the Weather MIDlet have been displayed using low-level MIDP graphics?

A Absolutely. Instead of using a form that contains a series of string items, you could have used a canvas and drawn the weather information on it as text. However, keep in mind that it would require a fair amount of calculation to position the text properly because there is no concept of multiple lines of text when drawing low-level graphics.

Q Is it possible to obtain weather information directly from a weather server without having to parse a Web page?

A Yes. There are weather servers out there that you can connect to and receive weather information directly without having to read and parse a Web page. However, the weather information obtained through such a server still adheres to some kind of format that you'll have to learn and address when extracting the weather data. The Stocker MIDlet you develop on Day 15, "Managing Your Finances," is a good example of a MIDlet that directly connects to a specialized server to retrieve data.

Q Is there a risk of the Weather.gov Web site changing the format of the weather conditions and causing a problem in the Weather MIDlet?

A Yes. The Weather MIDlet is entirely dependent on the format of the weather data contained in the hourly.html Web page for each state. If the Weather.gov Web designers decide to change the formatting of the data then the Weather MIDlet will have to be modified to extract the data properly.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What is it about the Weather.gov Web site that made it particularly well suited for use with the Weather MIDlet?
2. Why is it necessary to convert the two-letter state abbreviation before passing it into the `getConditions()` method in the Weather MIDlet?
3. What is the purpose of the `done` flag in the Weather MIDlet?
4. What happens if for some reason the weather information cannot be found for the city and state entered in the Weather MIDlet?

Exercises

1. Test out the Weather MIDlet using your home city and state, making sure that the data matches the actual conditions.
2. Modify the Weather MIDlet so that it includes the barometric pressure as part of the weather conditions.

DAY 11

Wireless Navigation

The real excitement behind wireless and mobile devices is the ability you have to access information on the Internet when you don't have access to a traditional wired connection. Not surprisingly, the types of information you might want to access from a mobile, wireless device differ somewhat from what you might want to access from your desktop computer. One such type of information is driving directions that tell you how to navigate from one location to another. Imagine being able to find your way through town from your mobile phone no matter where you are located. You probably already know that this information is readily available on the Web through a Web browser. The trick is to somehow access it with a MIDlet on a mobile device.

This lesson tackles the challenge of designing and developing a Directions MIDlet that enables you to enter two locations, and then display the driving directions from one to the other. This MIDlet utilizes an existing Web site to query for a Web page containing the driving directions, and it then extracts the directions from the page and displays them on a special MIDlet screen. Following are the topics covered in the lesson to assemble the MIDlet:

- Navigating with a little help from the Web
- Dissecting a driving directions Web page
- Developing the Directions MIDlet
- Taking the Directions MIDlet for a test drive

Using the Web to Get From A to B

The World Wide Web was so named because all of the documents within it form a massively complex web of interconnected data. Long before the Web entered the picture, there was a much more tangible "web" that we all still depend on a great deal. A web of roads is cut through the land all over the world. Although virtually everyone is dependent on roads in one way or another, the vast majority of us use them directly to get from one place to another and conduct business, purchase food, shuttle kids to baseball games, and so forth. And as much as we might be familiar with the roads we live near, a whole world of roads is out there just waiting for us to get lost in.

An interesting application of the Web is the sharing of maps that can be viewed to better understand how to travel by road more efficiently, and hopefully without getting lost. In addition to simply viewing maps on the Web, you can also obtain detailed driving directions that explain exactly how to get from one street address to another. If you've never used a Web site that provides this feature, I encourage you to try it because you'll be amazed at how accurate the results are. Figure 11.1 shows the MapBlast.com Web site, which is located at `http://www.mapblast.com/`.

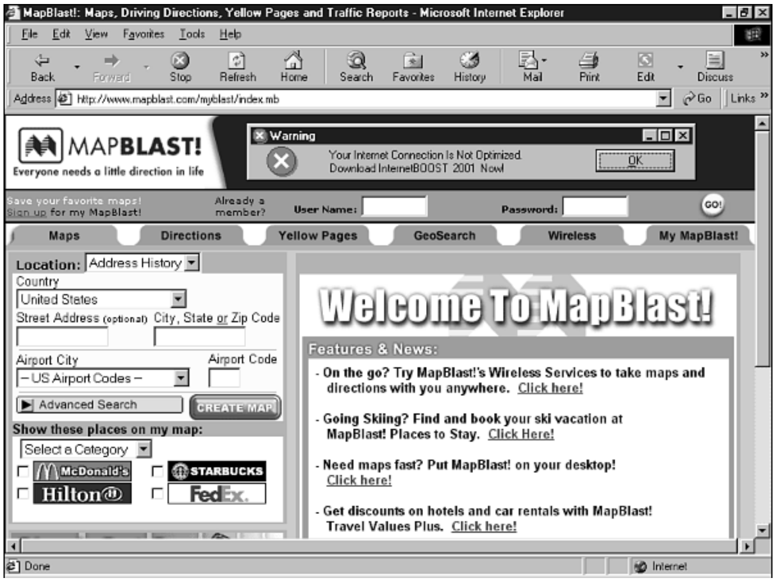
The MapBlast.com Web site is very powerful and provides access to a staggering amount of mapping information. To obtain driving directions from one street address to another, click the Directions tab on the MapBlast home page. Figure 11.2 shows the MapBlast Web page used to enter the two street addresses for retrieving driving directions.

After entering the From and To locations in the Directions tab of MapBlast, you can click the GetDirections button on the page to retrieve the driving directions. Figure 11.3 shows the resulting driving directions displayed by MapBlast.

As you can see, MapBlast returns a visual map that highlights the path from one location to the other. More applicable to this lesson are the text driving directions, which appear below the map and are made visible by scrolling down the page (see Figure 11.4).

FIGURE 11.1

The MapBlast.com Web site enables you to view maps and driving directions all over the world.


File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Go Links
MAPBLAST!
Everyone needs a little direction in life
Warning
OK
User Name: Password: GO!
Maps Directions Yellow Pages GeoSearch Wireless My MapBlast!
Advanced Search
CREATE MAP
McDonald's STARBUCKS Hilton FedEx
Welcome To MapBlast!
Done
Internet

MapBlast!: Maps, Driving Directions, Yellow Pages and Traffic Reports - Microsoft Internet Explorer

Address http://www.mapblast.com/myblast/index.mb

Your Internet Connection Is Not Optimized.
Download InternetBOOST 2001 Now!

Save your favorite maps! Sign up for my MapBlast! Already a member?

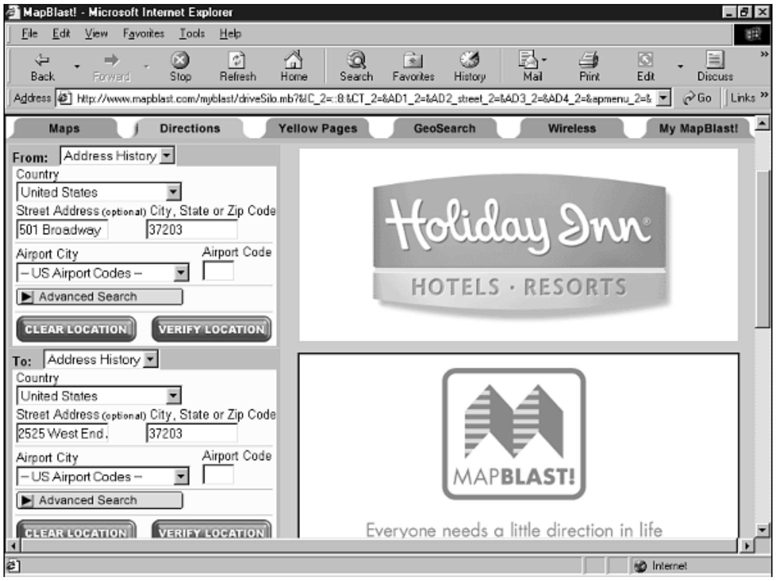
Location: Address History
Country: United States
Street Address (optional) City, State or Zip Code
Airport City: -- US Airport Codes -- Airport Code
Show these places on my map: Select a Category

Features & News:

- On the go? Try MapBlast!'s Wireless Services to take maps and directions with you anywhere. Click here!
- Going Skiing? Find and book your ski vacation at MapBlast! Places to Stay. Click Here!
- Need maps fast? Put MapBlast! on your desktop! Click here!
- Get discounts on hotels and car rentals with MapBlast! Travel Values Plus. Click here!

FIGURE 11.2

The Directions tab of the MapBlast.com Web site enables you to enter two locations and obtain the driving directions to get from one to the other.

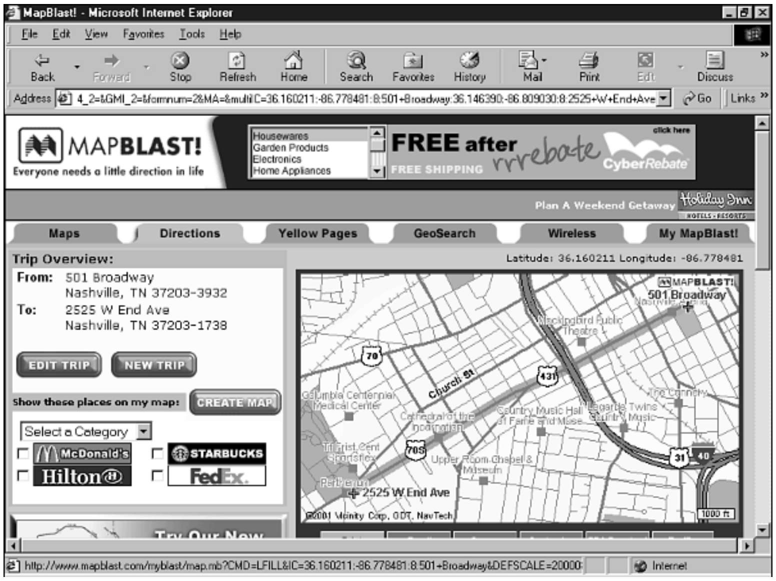

File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Go Links
Maps Directions Yellow Pages GeoSearch Wireless My MapBlast!
Advanced Search
CLEAR LOCATION VERIFY LOCATION
Holiday Inn HOTELS · RESORTS
MAPBLAST!
Everyone needs a little direction in life
Internet

MapBlast! - Microsoft Internet Explorer

Address http://www.mapblast.com/myblast/driveSilo.mb?&IC_2=:8:&CT_2=&AD1_2=&AD2_street_2=&AD3_2=&AD4_2=&apmenu_2=&

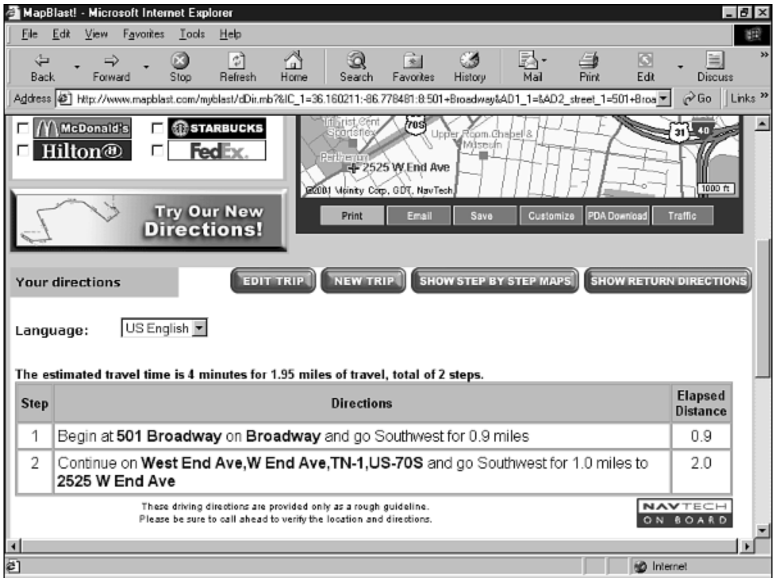
From: Address History
Country: United States
Street Address (optional): 501 Broadway City, State or Zip Code: 37203
Airport City: -- US Airport Codes -- Airport Code

To: Address History
Country: United States
Street Address (optional): 2525 West End. City, State or Zip Code: 37203
Airport City: -- US Airport Codes -- Airport Code


MapBlast! - Microsoft Internet Explorer
MAPBLAST!
Everyone needs a little direction in life
Maps
Directions
Yellow Pages
GeoSearch
Wireless
My MapBlast!
Trip Overview:
Latitude: 36.160211 Longitude: -86.778481
EDIT TRIP
NEW TRIP
Show these places on my map:
CREATE MAP
Select a Category
501 Broadway
2525 W End Ave

From: 501 Broadway
Nashville, TN 37203-3932
To: 2525 W End Ave
Nashville, TN 37203-1738

FIGURE 11.3
The driving directions provided by the MapBlast.com Web site are presented as a graphical map.



The estimated travel time is 4 minutes for 1.95 miles of travel, total of 2 steps.

Step	Directions	Elapsed Distance
1	Begin at **501 Broadway** on **Broadway** and go Southwest for 0.9 miles	0.9
2	Continue on **West End Ave,W End Ave,TN-1,US-70S** and go Southwest for 1.0 miles to **2525 W End Ave**	2.0

These driving directions are provided only as a rough guideline.
Please be sure to call ahead to verify the location and directions.

FIGURE 11.4
Scrolling down the MapBlast.com driving directions Web page reveals text directions.

The text driving directions are broken down into steps that explain each leg of the trip involved in driving from one location to the other. This is the type of information that you will be accessing later in the lesson when you develop the Directions MIDlet. If you look carefully at the text driving directions made available by MapBlast, you can see something that could prove problematic for the Directions MIDlet. The street names in the directions are in bold, which means that HTML tags (`<b>` and `</b>`) are being used to dress up the text. Although this makes the text look better on the MapBlast driving directions Web page, it adds unwanted code to the text that you must filter out before displaying it in a MIDlet.

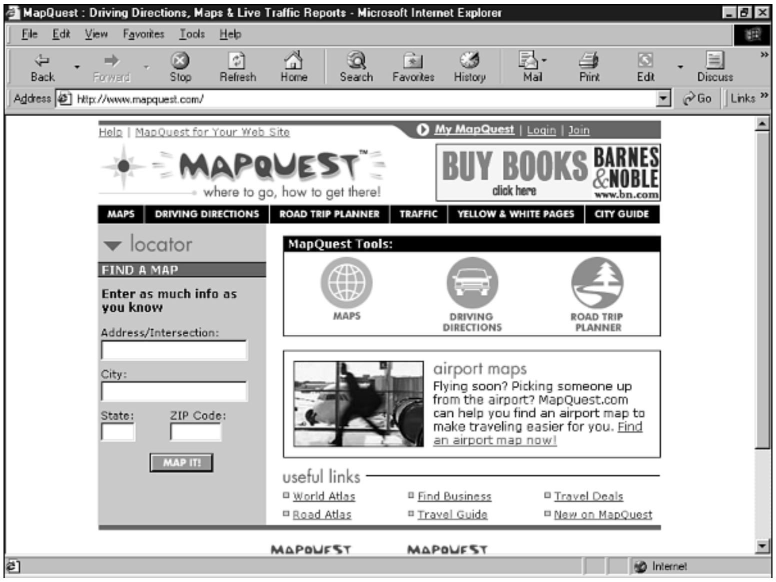
Fortunately, an easy solution is available to this problem. Another similar mapping Web site that generates driving directions a little easier to process is the MapQuest.com Web site. Figure 11.5 shows the MapQuest.com home page, which offers features similar to MapBlast.

Note

If MapQuest offers driving directions in a better format than MapBlast, why bother mentioning MapBlast? After you finish this book you will likely face similar challenges when it comes to finding information sources to feed your MIDlets. In this example, I analyzed two different potential data sources and decided on one of them for a very specific reason. To understand exactly why this decision was made, you need to see both Web sites.

FIGURE 11.5

Similar to MapBlast.com, the MapQuest.com Web site enables you to view maps and driving directions.

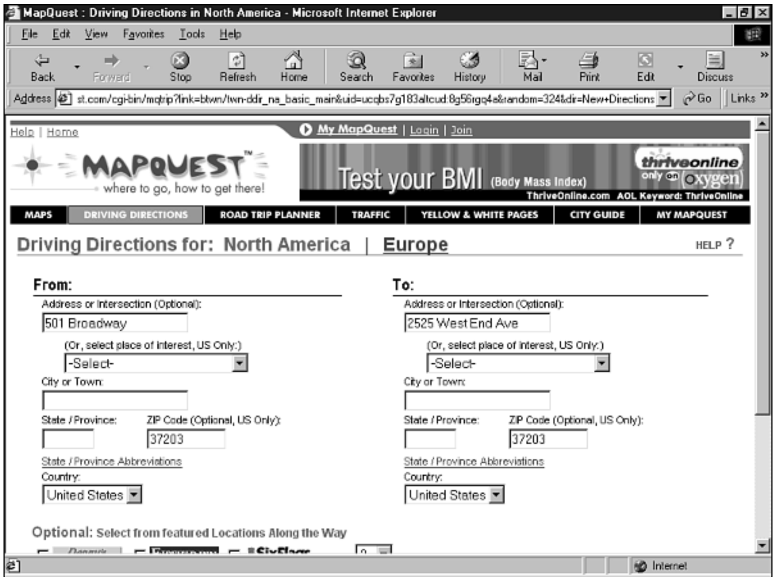

MapQuest : Driving Directions, Maps & Live Traffic Reports - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.mapquest.com/
Go Links
Help | MapQuest for Your Web Site
My MapQuest | Login | Join
MAPQUEST where to go, how to get there!
BUY BOOKS click here BARNES & NOBLE www.bn.com
MAPS DRIVING DIRECTIONS ROAD TRIP PLANNER TRAFFIC YELLOW & WHITE PAGES CITY GUIDE
locator
FIND A MAP
Enter as much info as you know
Address/Intersection:
City:
State: ZIP Code:
MAP IT!
MapQuest Tools:
MAPS DRIVING DIRECTIONS ROAD TRIP PLANNER
airport maps
Flying soon? Picking someone up from the airport? MapQuest.com can help you find an airport map to make traveling easier for you. Find an airport map now!
useful links
World Atlas Find Business Travel Deals
Road Atlas Travel Guide New on MapQuest
Internet

Like MapBlast, MapQuest enables you to enter two locations to retrieve driving directions to get from one to the other. To access this part of the MapQuest Web site, click the

Driving Directions tab near the top of the page. Figure 11.6 shows how the MapQuest driving directions Web page enables you to enter two street addresses for the From and To locations.

FIGURE 11.6

The Driving Directions tab of the MapQuest.com Web site enables you to enter two locations and obtain the driving directions to get from one to the other.


MapQuest : Driving Directions in North America - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Go Links
Help | Home
My MapQuest | Login | Join
MAPQUEST where to go, how to get there!
Test your BMI (Body Mass Index) thriveonline only on oxygen ThriveOnline.com AOL Keyword: ThriveOnline
MAPS DRIVING DIRECTIONS ROAD TRIP PLANNER TRAFFIC YELLOW & WHITE PAGES CITY GUIDE MY MAPQUEST
HELP ?
Internet

Address: st.com/cgi-bin/mqtrip?link=btwn/twn-ddir_na_basic_main&uid=ucqbs7g183altcud:8g56igq4a&random=324&dir=New+Directions

Driving Directions for: North America | Europe

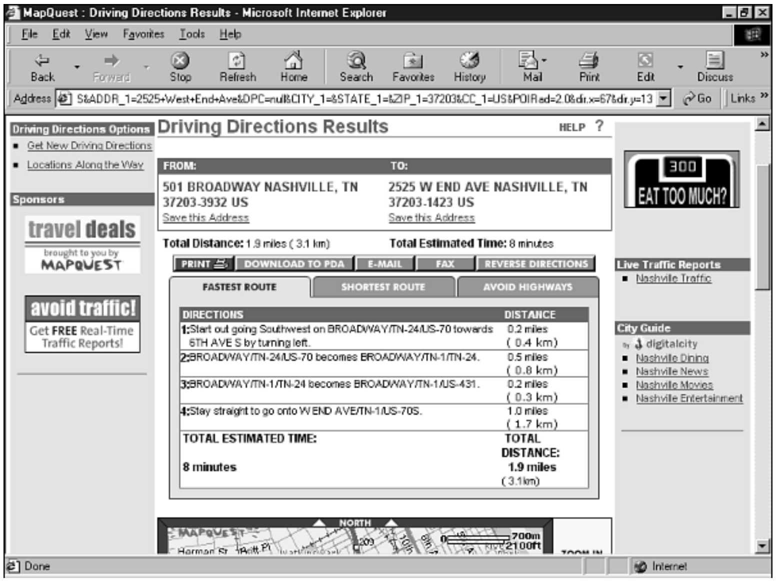
From:
Address or Intersection (Optional): 501 Broadway
(Or, select place of interest, US Only:) -Select-
City or Town:
State / Province:
ZIP Code (Optional, US Only): 37203
State / Province Abbreviations
Country: United States

To:
Address or Intersection (Optional): 2525 West End Ave
(Or, select place of interest, US Only:) -Select-
City or Town:
State / Province:
ZIP Code (Optional, US Only): 37203
State / Province Abbreviations
Country: United States

Optional: Select from featured Locations Along the Way

After you've entered two street addresses, the driving directions are easily obtained by clicking the Get Directions button near the bottom of the page. The resulting directions are shown in Figure 11.7.

As you can see in Figure 11.7, the driving directions provided by MapQuest are first presented as text and then followed by a graphical image. Unlike the MapBlast text directions, the MapQuest text is completely unformatted, which happens to be exactly what we need to extract data for the Directions MIDlet. Now you just need to learn how to dynamically retrieve a Web page with driving directions for two locations, and then take a look at the structure of the HTML code for the page to see how to extract the directions themselves.



MapQuest : Driving Directions Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address S&ADDR_1=2525+West+End+Ave&DPC=null&CITY_1=&STATE_1=&ZIP_1=37203&CC_1=US&POIRad=2.0&dir.x=67&dir.y=13
Go Links
Driving Directions Options
Get New Driving Directions
Locations Along the Way
Sponsors
travel deals brought to you by MAPQUEST
avoid traffic! Get FREE Real-Time Traffic Reports!
HELP ?
EAT TOO MUCH?
Live Traffic Reports
Nashville Traffic
City Guide
digitalcity
Nashville Dining
Nashville News
Nashville Movies
Nashville Entertainment
PRINT
DOWNLOAD TO PDA
E-MAIL
FAX
REVERSE DIRECTIONS
FASTEST ROUTE
SHORTEST ROUTE
AVOID HIGHWAYS
NORTH
Done
Internet

Driving Directions Results

FROM:	TO:
501 BROADWAY NASHVILLE, TN 37203-3932 US Save this Address	2525 W END AVE NASHVILLE, TN 37203-1423 US Save this Address

Total Distance: 1.9 miles (3.1 km) Total Estimated Time: 8 minutes

DIRECTIONS	DISTANCE
1:Start out going Southwest on BROADWAY/TN-24/US-70 towards 6TH AVE S by turning left.	0.2 miles (0.4 km)
2:BROADWAY/TN-24/US-70 becomes BROADWAY/TN-1/TN-24.	0.5 miles (0.8 km)
3:BROADWAY/TN-1/TN-24 becomes BROADWAY/TN-1/US-431.	0.2 miles (0.3 km)
4:Stay straight to go onto W END AVE/TN-1/US-70S.	1.0 miles (1.7 km)
TOTAL ESTIMATED TIME: 8 minutes	TOTAL DISTANCE: 1.9 miles (3.1km)

FIGURE 11.7

The driving directions provided by the MapQuest.com Web site are presented as text followed by a graphical map.

Ripping Apart a Driving Directions Web Page

To build a MIDlet capable of dynamically extracting driving directions from a Web page, you must first understand how such a Web page is generated. By reverse-engineering the URL of the MapQuest Web pages created when obtaining driving directions, it becomes obvious that parameters are used directly in the URL to identify the two locations involved in the driving directions. In other words, all of the information required to generate driving directions is placed within one large URL. This means that you can generate driving directions for any two locations by carefully constructing the URL programmatically.

Following is a sample URL used to retrieve driving directions with MapQuest:

```
http://www.mapquest.com/cgi-bin/mqtrip?
link=btwn/twn-ddir_options_jumppage&print.x=1&
ADDR_0=501%20broadway&ZIP_0=37203&
ADDR_1=2525%20west%20end%20ave&ZIP_1=37203&
CC_0=US&CC_1=US
```

I've deliberately broken the URL at certain points so that it is a little easier to read. However, if you were to type the above URL into a Web browser you would get the driving directions to take you from the Gaylord Entertainment Center to P.F. Chang's China Bistro, both in Nashville, TN. I chose these two places because the directions describe a pretty good night on the town—a Nashville Predators hockey game followed by a great meal! The point to the discussion is that the URL contains parameters that specify each of the pieces of information required to retrieve the driving directions. The following are these parameters and what they mean:

- **`ADDR_0`**—The street address to get directions from
- **`ZIP_0`**—The ZIP code to get directions from
- **`ADDR_1`**—The street address to get directions to
- **`ZIP_1`**—The ZIP code to get directions to
- **`CC_0`**—The country code to get directions from
- **`CC_1`**—The country code to get directions to

These parameters are all very straightforward, and describe the From and To locations involved in the driving directions. Notice a few things about the formatting of these parameters, however. If you notice, each parameter in the URL is divided by an ampersand (`&`). So, it is important to use an ampersand to separate parameters. Additionally, any spaces that would normally appear in a street address are replaced by the character code `%20`. The reason for this is that URLs aren't exactly designed to allow spaces, so it is necessary to use this special character to delineate street numbers and words in street addresses without actually using space characters. Knowing this, you can surmise that you'll need to account for the conversion of spaces in the Directions MIDlet because the user will certainly be entering street addresses with spaces.

Note

> Nothing is magical about the `%20` character that is used as a separator in the driving directions URL—it just serves as a means of distinguishing a space without actually using a space character.

You now have an idea about how to obtain a Web page containing driving directions between two specific locations. This means you can safely turn your attention to the data contained in the Web page itself. Keep in mind that the Directions MIDlet will be reading this data and somehow extracting driving directions from the HTML code. Sound difficult? Maybe so, but let's take a look at an excerpt of HTML code for a driving directions Web page to see how tough a challenge this is going to be (Listing 11.1).

LISTING 11.1 An Excerpt of HTML Code for a Driving Directions Web Page

```
<!-- connectlib 4.6 connect_4.6_mqlib6.4.2_SR051000_18:17:50_MDT_05/10/2000 -->
<! Opening ../../html/btwn/twn-ddir_options_jumppage.html >
<! Opening ../../html/btwn/twn-ddir_print.html >
<! Opening ../../html/bglobal/ddir-print.html >

<html>
<head>
<title>MapQuest&nbsp;:&nbsp;Printer-Friendly Driving Directions</title>
<link rel="stylesheet" type="text/css" href="/mq.css/twn.css">
<style type="text/javascript" src="/mq.css/twn.jss"></style>
</head>
<body bgcolor="#ffffff" leftmargin="0" topmargin="0" marginwidth="0"
➥marginheight="0">
<center>

<table cellpadding="2" cellspacing="0" width="620" border="0">
<tr><td colspan=2 align=center valign=top>
<img src="http://mqgraphics.mapquest.com/gif/mq-print_logo.gif"
➥width=298 height=54 border=0
➥ alt="MapQuest.com" vspace=5></td></tr>
  <tr>
    <td valign=bottom><font face="Arial, Helvetica, sans-serif"
➥size="4" color="#666666">
➥<b>Driving Directions Results</b></font></td>
<td align="right">
<div align=right class="leftColumn" width="115">
<a href="javascript:history.go(-1)"><font color="#808080">
➥Back</font></a> &nbsp;&nbsp;
<a href="javascript:window.print();"><img
➥src="http://mqgraphics.mapquest.com/gif/sendtoprinter.gif" width="115"
➥height="18" border="0" alt="Send To Printer"></a></div>
</td>
  </tr>
</table>

<table border=0 cellpadding=1 cellspacing=0 width=620 bgcolor=#666666>
<tr>
<td width="310" class="leftHead">&nbsp;FROM:</td>
    <td width="310" class="leftHead">&nbsp;TO:</td>
</tr>
<tr><td colspan=2 width="620">
<table cellpadding="5" cellspacing="0" width="100%" border="0" bgcolor=#ffffff>
<tr>
 <td valign=top width=310 class="formType">
501 BROADWAY
NASHVILLE, TN
37203-3932
US
```

LISTING 11.1 continued

```
</td>
 <td valign=top width=310 class="formType">
2525 W END AVE
NASHVILLE, TN
37203-1423
US
</td>
</tr>
</table></td></tr>
</table>

<table border=0 cellpadding=1 cellspacing=0 width=620 bgcolor=#666666>
<tr><td>
<table border=0 cellpadding=0 cellspacing=0 width=100% bgcolor=#ffffff>


<!-- Titles -->
<tr bgcolor=#666666>
<!-- Only show the thumbnails if we're displaying turn-by-turn maps -->

<td class="leftHead">&nbsp;DIRECTIONS</td>
<td class="leftHead">&nbsp;DISTANCE</td>
</tr>

<!-- Content -->
  <!-- Displays instructions if origin cannot be routed from -->

<tr>
  <!-- Only show the thumbnails if we're displaying turn-by-turn maps -->
<td valign=top>
<font face=arial,helvetica size=2 color=#000000>
<b>1:</b>
Start out going Southwest on BROADWAY/TN-24/US-70 towards 6TH AVE
➥S by turning left.
</font></td>
<td valign=top>
<font face=arial,helvetica size=2 color=#000000>
&nbsp;
     0.2 miles
</font>
<br>
<font face=arial,helvetica size=1 color=#000000>
&nbsp;
(      0.4 km)
</font></td>
</tr>
<tr>
     <td colspan=2 bgcolor="#666666"><img
```

LISTING 11.1 continued

```
➥src="http://mqgraphics.mapquest.com/gif/blank_1x1.gif" height="1"
➥width="1" alt="" border="0"></td>
</tr>
<! $GEO_LOOP END maneuver_poi_nested_loop>
<tr>
  <!-- Only show the thumbnails if we're displaying turn-by-turn maps -->
<td valign=top>
<font face=arial,helvetica size=2 color=#000000>
<b>2:</b>
BROADWAY/TN-24/US-70 becomes BROADWAY/TN-1/TN-24.
</font></td>
```

Note

A typical driving directions Web page generated by MapQuest is considerably longer than the example shown here. I only list enough code to give you an idea about how a typical page is laid out.

The HTML code for this page looks like chaos, right? I agree, but if you were to study the code closely you would notice an important pattern with respect to the text driving directions. Because the directions are listed as numeric steps, each step is preceded by a line of code with the following form:

```
<b>X:</b>
```

The *X* in this code is the step number. So, for example, the HTML code preceding the first step of the driving directions looks like this:

```
<b>1:</b>
```

Similarly, the second step, which appears near the bottom of the previous listing, is preceded by this code:

```
<b>2:</b>
```

This is precisely the type of pattern you need to drill down into the HTML code and cleanly rip out each step of the driving directions. However, what you don't know going into each Web page is the number of total steps. So, the solution is to read each numbered step in succession until you run out of them, and then you know the directions are over. This can sound like it's easier said than done, but you find out in the next section that the process of reading and extracting driving directions from the MapQuest.com Web site is not difficult. Of course, this makes the Directions MIDlet dependent on the format of the MapQuest directions page. You will have to update the MIDlet if MapQuest ever changes the format of their directions pages.

Constructing the Directions MIDlet

Similar to the Weather MIDlet that you built in the preceding lesson, the Directions MIDlet relies on the extraction of information from a Web page that is generated on-the-fly. The Directions MIDlet is responsible for building a URL that results in the generation of this page, as well as reading data from the page and parsing out a list of driving directions. The Directions MIDlet is also similar to the Weather MIDlet in that it must utilize two user interfaces: one that enables the user to enter the From and To locations, and one to display the resulting driving directions.

The developmental tasks for the Directions MIDlet closely parallel those of the Weather MIDlet. For this reason, you'll develop the Directions MIDlet using a similar approach as was used for the Weather MIDlet. More specifically, the development of the Directions MIDlet can be broken down into the following familiar parts:

- The user interface
- Handling commands
- Retrieving weather data

The next few sections guide you through the design and coding of each of these facets of the Directions MIDlet, along with a wrap-up where you see it all come together.

The User Interface

The function of the Directions MIDlet is to prompt the user to enter two locations and use that information to retrieve driving directions from the Web. After the driving directions have been obtained, they must be presented to the user. This means that the MIDlet consists of two different screens: one to get the location input from the user and another to display the actual driving directions after they have been successfully processed and retrieved.

The Directions MIDlet is designed only to support locations within the United States, which means that only four pieces of information are required from the user: From street address, From ZIP code, To street address, and To ZIP code. This information can be obtained through text-field GUI components placed on an input screen. After entering the information into text fields on the input screen, the user selects the `Go` command to initiate the retrieval of the driving directions. At this point a different screen is displayed that contains the driving directions. Similar to the Weather MIDlet, the Directions MIDlet is always displaying one of the two screens, which means the screens together form the MIDlet's user interface.

The first screen handles accepting the user's input for the two locations, so it makes sense to refer to this screen as the locations screen. Because this screen needs to hold four text fields for the different pieces of location information, it is best implemented as a form. The second screen, known as the directions screen, must also be a form because it will present each step of the driving directions as string-item GUI components.

As with any MIDlet, the Directions MIDlet requires a few commands in addition to its screens. The Directions MIDlet functions similarly to the Weather MIDlet because it uses the locations screen as the main screen with `Exit` and `Go` commands. Likewise, the directions screen serves as the output screen and has only the single `Back` command. The screens and commands for the MIDlet are declared as member variables near the top of the Directions class, as the following code shows:

```
private Command exitCommand, goCommand, backCommand;
private Display display;
private Form locationsScreen;
private TextField addrFromField, zipFromField, addrToField, zipToField;
private Form directionsScreen;
```

The command variables are declared first, and are familiar as the `Exit`, `Go`, and `Back` commands, respectively. The very familiar `Display` object is then declared. The `locationsScreen` variable is declared as a `Form` object, which is necessary because it will contain the text fields used to enter the location information. Text fields are declared as `TextField` objects, and enable the user to enter the street address and ZIP code for the From and To locations. The last member variable is `directionsScreen`, which is declared as a `Form` object. Even though the `directionsScreen` variable is declared as a form, notice that no other GUI component member variables are declared. This is because the string items that will reside in the directions screen are created dynamically as needed. The reason for this has to do with the fact that you don't know how many steps are in the driving directions until later.

These member variables are all initialized in the MIDlet's constructor, `Directions()`, whose code is shown in Listing 11.2.

Listing 11.2 The `Directions()` Constructor Initializes the Fortune MIDlet

```
public Directions() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the Exit, Go, and Back commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  goCommand = new Command("Go", Command.OK, 2);
  backCommand = new Command("Back", Command.BACK, 2);
```

LISTING 11.2 continued

```
    // Create the locations screen form
    locationsScreen = new Form("Enter Locations");
    addrFromField = new TextField("From Address", "501 broadway", 30,
➥TextField.ANY);
    locationsScreen.append(addrFromField);
    zipFromField = new TextField("From Zip Code", "37203", 5,
➥TextField.NUMERIC);
    locationsScreen.append(zipFromField);
    addrToField = new TextField("To Address", "2525 west end ave", 30,
➥TextField.ANY);
    locationsScreen.append(addrToField);
    zipToField = new TextField("To Zip Code", "37203", 5, TextField.NUMERIC);
    locationsScreen.append(zipToField);

    // Set the Exit and Go commands for the locations screen
    locationsScreen.addCommand(exitCommand);
    locationsScreen.addCommand(goCommand);
    locationsScreen.setCommandListener(this);

    // Create the directions screen form
    directionsScreen = new Form("Directions");

    // Set the Back command for the directions screen
    directionsScreen.addCommand(backCommand);
    directionsScreen.setCommandListener(this);
  }
```

The first chore in the constructor is to obtain a `Display` object, which is a standard task required of most MIDlets. The three commands used throughout the MIDlet (`Exit`, `Go`, and `Back`) are then created. So far the code still looks very similar to the constructor code in the Weather MIDlet. However, things change a little as you get into the creation of the locations screen. The screen is created as a `Form` object, along with its four `TextField` components. The components are all initialized with default To and From addresses, which make it easy to test the MIDlet without having to do much typing on the device keypad. The components are added to the screen through calls to the `append()` method. Because the `Exit` and `Go` commands are associated with the locations screen, the `exitCommand` and `goCommand` objects are added to the screen as commands by calling the `addCommand()` method.

The initialization of the directions screen is quite simple given the fact that it doesn't hold any components yet. After creating the `Form` object, the `backCommand` object is added to the screen to establish the `Back` command. That's all the work necessary for the directions screen at this point. You're now ready to move on to the command handler code for the MIDlet.

Handling Commands

As you are well aware, the Directions MIDlet includes three commands: `Exit`, `Go`, and `Back`. The method responsible for handling these commands is the `commandAction()` method, whose code is shown in Listing 11.3.

LISTING 11.3 The `commandAction()` Method Handles Commands for the Directions MIDlet

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == goCommand) {
    // Get the directions
    getDirections(addrFromField.getString().toLowerCase(),
      zipFromField.getString().toLowerCase(),
      addrToField.getString().toLowerCase(),
      zipToField.getString().toLowerCase());
  }
  else if (c == backCommand) {
    // Set the current display back to the locations screen
    display.setCurrent(locationsScreen);

    // Clear the directions
    for (int i = directionsScreen.size() - 1; i >= 0; i--)
      directionsScreen.delete(i);
  }
}
```

The `commandAction()` method tackles the `Exit` command first by comparing the command parameter (`c`) with the `exitCommand` member variable. If there is a match, the MIDlet is exited. The `Go` command is handled with a call to the `getDirections()` method, which accepts four string arguments containing the location information for which you would like directions. The method prototype for the `getDirections()` method reveals its use:

```
void getDirections(String addrFrom, String zipFrom,
  String addrTo, String zipTo)
```

The `getDirections()` method accepts the From address, From ZIP code, To address, and To ZIP code as string parameters, and then uses them to obtain driving directions that are displayed on the directions screen. All of the strings are first converted to lowercase to make the input data more consistent because the URL for the MapQuest Web page is being constructed from this information.

The last command handled in the `commandAction()` method is the `Back` command, which first sets the screen back to the locations screen, and then clears the string items from the directions screen. Unlike the Weather MIDlet, which clears the city and state each time you go back to the location screen, the Directions MIDlet maintains the two locations entered on the locations screen in case the user wants to change them and try again. Users can also exit the MIDlet from the locations screen if they are finished getting directions.

Prepping the Location Information

Earlier in this lesson I mentioned that the URL for obtaining the driving directions Web page required spaces to be converted to a special character (`%20`). The reason for this is that URLs aren't designed to accommodate spaces, so you are effectively masking spaces with the `%20` character. This is important for the Directions MIDlet because the user will be entering street addresses with spaces that must be removed when constructing the URL for MapQuest. To make it easy to replace these pesky spaces with `%20` characters, it is helpful to create a method that accepts a string parameter and replaces any spaces within the string. Listing 11.4 contains the code for such a method.

LISTING 11.4 The `replaceSpaces()` Method Replaces Spaces in a String with `%20` Characters

```
private String replaceSpaces(String s) {
  StringBuffer str = new StringBuffer(s);

  // Read each character and replace if it's a space
  for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) == ' ') {
      str.deleteCharAt(i);
      str.insert(i, "%20");
    }
  }

  return str.toString();
}
```

The `replaceSpaces()` method takes a string as its only parameter and returns another string that is the original string with spaces replaced by the `%20` character. The method begins by converting the string to a string buffer, which enables you to delete and insert characters within the string. A `for` loop is then entered that iterates through the string a character at a time. Any time a space is encountered in the string of text, it is deleted and a `%20` character is inserted in its place. This method will prove to be very handy in the next section when you assemble the URL for the driving directions Web page.

Retrieving Directions Data

The real work of the Directions MIDlet is the retrieval of the actual driving directions. This involves creating a URL, opening a network connection, reading the Web page, parsing out the driving directions, and displaying the directions on the directions screen. Sound tough? Not really! In fact, you can look to the `getConditions()` method of the Weather MIDlet as a rough template for how the `getDirections()` method needs to be structured. Before you get into the code, however, let's review exactly what tasks are being carried out in the `getDirections()` method:

1. Construct a URL for the driving directions Web page.
2. Open a network connection to the URL.
3. Read data from the network connection and search for a line of code with the form `<b>`*`X`*`:</b>`.
4. When a match is found, read the next line as a step of the directions.
5. Create a string item on the directions screen and set its value to the current step text.
6. Set the current display to the directions screen.

Hopefully these steps help to clarify the role of the `getDirections()` method within the Directions MIDlet. At this point You will better understand the method by diving into its code, which is shown in Listing 11.5.

LISTING 11.5 The `getDirections()` Method Obtains Driving Directions from a MapQuest Directions Web Page

```
private void getDirections(String addrFrom, String zipFrom,
  String addrTo, String zipTo) {
  StreamConnection conn = null;
  InputStream in = null;
  StringBuffer data = new StringBuffer();

  // Replace any spaces in the addresses
  addrFrom = replaceSpaces(addrFrom);
  addrTo = replaceSpaces(addrTo);

  // Build the URL for the directions page
  String url = "http://www.mapquest.com/cgi-bin/mqtrip?link=btwn
➥/twn-ddir_options_jumppage&" +
    "print.x=1&" + "ADDR_0=" + addrFrom + "&ZIP_0=" + zipFrom +
    "&ADDR_1=" + addrTo + "&ZIP_1=" + zipTo + "&CC_0=US&CC_1=US";
```

LISTING 11.5 continued

```
    try {
      // Open the HTTP connection
      conn = (StreamConnection)Connector.open(url);

      // Obtain an input stream for the connection
      in = conn.openInputStream();

      // Read a line at a time from the input stream
      int ch, step = 1;
      boolean isStep = false;
      while ((ch = in.read()) != -1) {
        if (ch != '\n') {
          // Read the line a character at a time
          data.append((char)ch);
        }
        else {
          // If this line is a step, add it to the directions text box
          if (isStep) {
            directionsScreen.append(new StringItem("", "* " + data.toString()));
            isStep = false;
          }
          else
            // See if this line precedes a step in the directions
            if (data.toString().compareTo("<b>" + step + ":</b>") == 0) {
              isStep = true;
              step++;
            }

          // Clear the string for the next line
          data = new StringBuffer();
        }
      }
    }
    catch (IOException e) {
      System.err.println("The connection could not be established.");
    }

    // Display the directions screen if everything is OK, otherwise display an
  alert
    if (directionsScreen.size() > 0)
      display.setCurrent(directionsScreen);
    else
      display.setCurrent(new Alert("Directions",
        "The locations are invalid. Please try again.", null, AlertType.ERROR));
  }
```

The first code in the `getConditions()` method creates several member variables that are used throughout the method. The `conn` variable is a `StreamConnection` object that stores the stream connection for the driving directions Web page. The `in` variable is an `InputStream` object that is used to read from the Web page. Last, but not least, is the `data` variable, which is a string buffer that is used to store one line of the Web page at a time as it is being read.

The first major step in the `getConditions()` method is to construct the URL and open a network connection to it. Before you can do that, however, you must process the To and From location strings so that any spaces are converted to `%20` characters. This is accomplished by calling the `replaceSpaces()` method and passing in each string.

With the address strings properly formatted, you're ready to put together the URL for the driving directions Web page. Using the MapQuest URL parameters covered earlier in the lesson, this step is relatively straightforward. You assemble a large string that includes the To and From location information in the appropriate places. After the URL is ready, the static `open()` method of the `Connector` class is used to establish a network connection to the URL. This connection is cast to a `StreamConnection` object, which is then used as the basis for opening an input stream. The interesting code in the `getDirections()` method is located within the `while` loop that reads a character at a time from the input stream.

The `while` loop in the `getConditions()` method is designed to read the entire driving directions Web page one character at a time. Within the loop, each character is tested to see whether it is a newline (`'\n'`). This is done to see whether the end of a line has been reached, in which case you need to see if it precedes a step in the directions. You determine this by seeing if the line matches the format `<b>`*`X`*`:</b>`. If so, the `isStep` flag is set to `true` and the step number (`step`) is incremented. If not, then the `data` string buffer is cleared to make room for the next line of the Web page.

I skipped over the section of code that first checks to see whether `isStep` is set to `true`. If so, you know that the current line is a step of the driving directions, so you need to display it on the directions screen. Do this by creating a `StringItem` object and adding it to the directions screen. Of course, the text value of the string item is set to the step of the directions. After creating and adding the string item, the `isStep` flag is set to `false` so that the `while` loop will keep checking for more steps.

After the `while` loop exits, you can check and see whether the driving directions were successfully retrieved by looking at the size of the directions screen. This works because string items are only added to the screen if direction steps are found. If no directions were found then the directions screen will not contain any components—its size will be zero. If the size of the directions screen is greater than zero, the directions screen is displayed. If not, an error message is displayed with an `Alert` object.

That concludes the `getDirections()` method, which you hopefully were able to follow and understand. This method presents a unique approach to solving the challenge of extracting information from the Web for use within a MIDlet.

Putting It All Together

The `getDirections()` method is by far the most important method in the Directions MIDlet. In addition to this method, you've also learned about the `replaceSpaces()` utility method, the `commandAction()` command handler method, and the `Directions()` constructor. What you haven't seen is how all this code fits together to form the complete Directions MIDlet. Listing 11.6 contains the complete code for the MIDlet, which will hopefully help give you an idea about how everything comes together.

LISTING 11.6 The Complete Source Code for the Directions MIDlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
import javax.microedition.io.*;

public class Directions extends MIDlet implements CommandListener {
  private Command exitCommand, goCommand, backCommand;
  private Display display;
  private Form locationsScreen;
  private TextField addrFromField, zipFromField, addrToField, zipToField;
  private Form directionsScreen;

  public Directions() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit, Go, and Back commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    goCommand = new Command("Go", Command.OK, 2);
    backCommand = new Command("Back", Command.BACK, 2);

    // Create the locations screen form
    locationsScreen = new Form("Enter Locations");
    addrFromField = new TextField("From Address", "501 broadway", 30,
➥ TextField.ANY);
    locationsScreen.append(addrFromField);
    zipFromField = new TextField("From Zip Code", "37203", 5,
➥TextField.NUMERIC);
    locationsScreen.append(zipFromField);
    addrToField = new TextField("To Address", "2525 west end ave", 30,
➥ TextField.ANY);
```

LISTING 11.6 continued

```
    locationsScreen.append(addrToField);
    zipToField = new TextField("To Zip Code", "37203", 5, TextField.NUMERIC);
    locationsScreen.append(zipToField);

    // Set the Exit and Go commands for the locations screen
    locationsScreen.addCommand(exitCommand);
    locationsScreen.addCommand(goCommand);
    locationsScreen.setCommandListener(this);

    // Create the directions screen form
    directionsScreen = new Form("Directions");

    // Set the Back command for the directions screen
    directionsScreen.addCommand(backCommand);
    directionsScreen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the locations screen
    display.setCurrent(locationsScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == goCommand) {
      // Get the directions
      getDirections(addrFromField.getString().toLowerCase(),
        zipFromField.getString().toLowerCase(),
        addrToField.getString().toLowerCase(),
        zipToField.getString().toLowerCase());
    }
    else if (c == backCommand) {
      // Set the current display back to the locations screen
      display.setCurrent(locationsScreen);

      // Clear the directions
      for (int i = directionsScreen.size() - 1; i >= 0; i--)
        directionsScreen.delete(i);
    }
  }
```

LISTING 11.6 continued

```
private String replaceSpaces(String s) {
  StringBuffer str = new StringBuffer(s);

  // Read each character and replace if it's a space
  for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) == ' ') {
      str.deleteCharAt(i);
      str.insert(i, "%20");
    }
  }

  return str.toString();
}

private void getDirections(String addrFrom, String zipFrom,
  String addrTo, String zipTo) {
  StreamConnection conn = null;
  InputStream in = null;
  StringBuffer data = new StringBuffer();

  // Replace any spaces in the addresses
  addrFrom = replaceSpaces(addrFrom);
  addrTo = replaceSpaces(addrTo);

  // Build the URL for the directions page
  String url = "http://www.mapquest.com/cgi-bin/mqtrip?link=btwn/
➥twn-ddir_options_jumppage&" +
    "print.x=1&" + "ADDR_0=" + addrFrom + "&ZIP_0=" + zipFrom +
    "&ADDR_1=" + addrTo + "&ZIP_1=" + zipTo + "&CC_0=US&CC_1=US";

  try {
    // Open the HTTP connection
    conn = (StreamConnection)Connector.open(url);

    // Obtain an input stream for the connection
    in = conn.openInputStream();

    // Read a line at a time from the input stream
    int ch, step = 1;
    boolean isStep = false;
    while ((ch = in.read()) != -1) {
      if (ch != '\n') {
        // Read the line a character at a time
        data.append((char)ch);
      }
      else {
```

LISTING 11.6 continued

```
          // If this line is a step, add it to the directions text box
          if (isStep) {
            directionsScreen.append(new StringItem("", "* " + data.toString()));
            isStep = false;
          }
          else
            // See if this line precedes a step in the directions
            if (data.toString().compareTo("<b>" + step + ":</b>") == 0) {
              isStep = true;
              step++;
            }

          // Clear the string for the next line
          data = new StringBuffer();
        }
      }
    }
    catch (IOException e) {
      System.err.println("The connection could not be established.");
    }

    // Display the directions screen if everything is OK, otherwise
➥display an alert
    if (directionsScreen.size() > 0)
      display.setCurrent(directionsScreen);
    else
      display.setCurrent(new Alert("Directions",
        "The locations are invalid. Please try again.", null, AlertType.ERROR));
  }
}
```

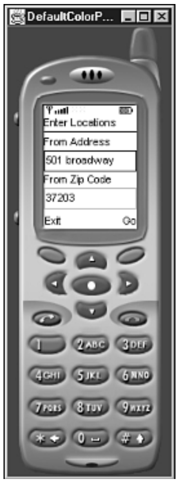
I didn't want to leave out some significant portion of the Directions MIDlet, so I have given the complete code listing. Keep in mind that although this code seems to be on the lengthy side, it's really not so bad when you consider what it is accomplishing. It would take much more effort to use a printed Road Atlas and come up with driving directions on your own!

Testing the Directions MIDlet

In general, the most fun part of any MIDlet project is the testing phase, provided there aren't too many bugs to hunt down. Fortunately, in this case you're dealing with a MIDlet that is already bug free (at least I hope it is), so the bug issue shouldn't enter the picture. As with any MIDlet, the Directions MIDlet must be compiled, pre-verified, and packaged to test it under the J2ME emulator. After that's been done, you can unleash the MIDlet in the emulator and go about retrieving driving directions all across the United States. Figure 11.8 shows the Directions MIDlet running in the emulator.

FIGURE 11.8

The Directions MIDlet prompts you to first enter the From location for obtaining driving directions.



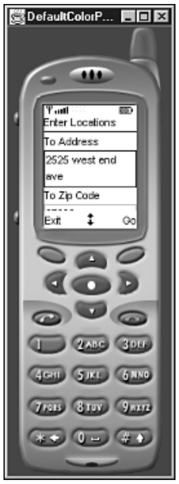
The first step in obtaining driving directions is to enter the From street address and ZIP code on the locations screen. After entering this, you can scroll down the screen to enter the To address and ZIP code, as shown in Figure 11.9.

Note

Use the Up and Down arrow buttons on the device to scroll up and down on a form.

FIGURE 11.9

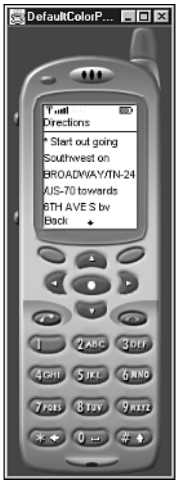
Scrolling down in the locations screen takes you to the To location for the driving directions.



After entering the To street address and ZIP code, you're ready to obtain the driving directions by selecting the `Go` command. Figure 11.10 shows the result of issuing this command on the default To and From locations in the Directions MIDlet.

FIGURE 11.10

After entering the From and To locations, and selecting the Go command, the Directions MIDlet displays the detailed driving directions.



```
Directions
* Start out going
Southwest on
BROADWAY/TN-24
/US-70 towards
6TH AVE S by
Back
```

As the figure shows, the Directions MIDlet displays the driving directions between the two locations as a list of text steps with each step preceded by an asterisk (`*`). You can scroll up and down in the directions using the Up and Down arrow keys. To return to the locations screen, select the `Back` command. From there you can enter a new pair of locations or exit the MIDlet.

Summary

This lesson guided you through the design and development of a highly practical MIDlet that can serve to get you out of trouble if you ever find yourself lost with nothing but a Java-powered mobile device. The lesson introduced map Web sites that enable you to obtain driving directions. You learned about the format of the driving directions generated by these Web sites and how one particular Web site formats its data so that it is easily accessible. From there, you began developing the Directions MIDlet, which extracts driving directions from a dynamically generated Web page and then displays the directions. Finally, the lesson ended by taking the Directions MIDlet for a test spin.

This lesson and the preceding one focused on the development of a couple of practical networking MIDlets. The next lesson shifts gears a little by exploring MIDlet optimization. You find out why it is important to optimize MIDlet code, as well as some strategies and tricks you can use to make your MIDlets more efficient.

Q&A

Q Is it possible to display the graphical map that is made available by the MapQuest.com Web site?

A Yes and no. Yes, it is technically possible to retrieve the map data and render it to the screen in a MIDlet. But no, it is probably not a good idea given the display and networking limitations of most mobile devices. It would be virtually impossible to display an entire map at once, which means you would be scrolling around a large image. Although this isn't necessarily a bad thing, such a large image would likely take a while to transfer over the relatively low bandwidth of a mobile network connection. Text directions are incredibly quick to gather and easy to display.

Q Are there any other ways the driving directions could be presented to the user to make the MIDlet easier to use?

A Yes. In fact, I strongly considered developing the MIDlet so that each step of the driving directions is displayed on its own screen. To navigate through the directions you would use `Forward` and `Back` commands. This approach is more than likely an improvement over the way the Directions MIDlet currently displays the directions, but I'll leave it to you to make the improvements to the MIDlet.

Q Is there a risk of the MapQuest.com Web site changing the format of the driving directions and causing a problem in the Directions MIDlet?

A Yes. Similar to the Weather MIDlet that you developed in the Day 10, "Being Your Own Wireless Meteorologist," the Directions MIDlet is entirely dependent on the format of the driving directions made available by MapQuest. If MapQuest ever decides to change the formatting of the directions then the Directions MIDlet must be modified to extract the data properly.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. Why are the MapQuest driving directions more suitable for use with a MIDlet than the MapBlast directions?
2. How does the Directions MIDlet find the text for each step of the driving directions within the HTML code for the Web page?

3. Why are no components added to the directions screen when the rest of the MIDlet's user interface is initialized?
4. What is the purpose of the `replaceSpaces()` method?

Exercises

1. Test out the Directions MIDlet using your home address and the address of a national landmark such as the White House. Just in case you've never sent mail to the president, the street address of the White House is 1600 Pennsylvania Avenue NW, and the ZIP code is 20500.
2. Modify the Directions MIDlet so that it includes the distance of each step of the directions (in miles) in parentheses at the end of each step. Hint: The distance for each step is available on the driving directions Web page and appears after the text for the step. One way to extract the distance of each step is to count the number of lines between the step text and the distance while processing the HTML code.

DAY 12

Optimizing MIDlet Code

We have seen that MIDP devices are constrained in terms of how much memory and processing power they have at their disposal. Mobile wireless devices have come a long way in the past few years, but it is still unfair to compare the capabilities of MIDP devices to desktop or laptop computers. For this reason, you must design and code MIDlets with the limited capabilities of MIDP devices in mind. Fortunately, the MIDP API goes a long way toward guiding developers in creating efficient MIDlets because the API itself is optimized for mobile devices. However, this does not mean there aren't things you can do to optimize your MIDlets for maximum performance.

This lesson explores MIDlet optimization by presenting several tips and techniques to get maximum performance from your MIDlets. The optimization strategies you learn in this lesson include MIDP-specific optimizations as well as general Java coding optimizations that apply to MIDlet development. The following major topics are covered in this lesson:

- Understanding the fundamentals of MIDlet optimization
- Exploring different strategies for optimizing MIDlets
- Examining coding techniques for optimizing MIDlet Java code

MIDlet Optimization Basics

The vast majority of mobile devices have significantly less computing power than their desktop and laptop counterparts. This normally wouldn't be of much concern because such devices have traditionally been used only to perform relatively mundane tasks. However, J2ME ushers in the notion of developing powerful applications for mobile devices, which means you have to consider the limitations imposed by the limited memory and processing power of the devices. Fortunately, the architects of J2ME have gone a long way to take care of this issue.

As you learned on Day 5, "Working within the CLDC and MIDP APIs," the CLDC and MIDP APIs are carefully designed to accommodate the needs of wireless mobile devices without adding much overhead. More specifically, the MIDP API is a significantly scaled-down version of the J2SE API that includes just enough features to develop interesting MIDlets. The MIDP API represents a series of tough trade-offs between providing as many useful features as possible for MIDlet development without putting an undue strain on device resources. The MIDP API goes a long way toward helping you to create small, efficient MIDlets that don't tax the resources of MIDP devices too terribly much. However, much of the responsibility of developing efficient MIDlets is still in your hands.

The reality is that the Java programming language is extremely flexible. You can create bloated, inefficient code even in an efficient environment such as the one offered by J2ME. Additionally, even if your code is reasonably efficient, the design of your MIDlet could be inefficient and result in poor performance. For example, it could be that you are processing far more data than is ideal within the constraints of a MIDP device, or maybe your MIDlet performs complex computational tasks that are not suited to a device with limited processing power. Anyone can get carried away and attempt to incorporate into MIDlets the depth of features found in desktop applications, which is usually not a good idea.

The moral of the story here is that you should do everything in your power to create *optimized MIDlets*, which are MIDlets that are designed to be extremely efficient, resulting in a minimal strain on device resources. There are obviously limitations to how much you can optimize a MIDlet. Realistically, mobile network connections are usually relatively slow, so it is to be expected that MIDlets that are heavily dependent on networking are not going to be as responsive as non-networked MIDlets. Even so, you should attempt to minimize the amount of data being transferred over a mobile network connection to help reduce this effect.

NEW TERM *optimized MIDlet*—A MIDlet that is designed to be extremely efficient, thereby putting a minimal strain on device resources.

There are several facets to MIDlet optimization, some of which are considerably more important than others. The following are the three main aspects of a MIDlet that you should focus on when optimizing your MIDlets:

- Maintainability
- Size
- Speed

The next few sections explore each of these types of optimization in more detail.

Optimizing for Maintainability

The least important type of optimization for MIDlets is maintainability optimization, which involves taking steps to help make MIDlet code more manageable in the future. This type of optimization is usually geared toward the structure and organization of code, rather than modifications of the algorithms used in the code. In general, maintainability optimization involves studying a MIDlet and making changes to help other programmers understand and modify its code in the future.

NEW TERM *maintainability optimization*—An optimization technique that involves taking steps to help make MIDlet code more manageable in the future.

In many ways, maintainability optimization works against the other two types of optimization because it stresses the understanding and organization of code in a MIDlet over the size or performance of the code. For this reason, maintainability optimization doesn't rank very high on the list of important optimizations used by MIDlet developers. It is still important to organize your code, enforce some structure, and by all means document the code well, but don't let maintainability optimization become an overriding concern.

Optimizing for Size

Another type of MIDlet optimization is *size optimization*, which primarily involves making changes to code that result in smaller executable class files. The cornerstone of size optimization is code reuse, which comes in the form of inheritance for Java classes. Fortunately, good object-oriented design strategies naturally favor size optimization, so you will rarely need to go out of your way to perform this type of optimization. For example, it is just good design practice to place code that is reused a few times within a method. In this way, some degree of size optimization naturally takes place during the initial code development for MIDlets. Size optimization is important to MIDlets because it determines the MIDlets' memory requirements.

New Term *size optimization*—An optimization technique that involves making changes to MIDlet code that result in smaller executable class files.

Another facet of MIDlet size optimization deals with the resources required by MIDlets and the data created by MIDlets. In addition to a MIDlet being small in terms of its executable class files, it is important for the resources and data that it uses to be as small as possible. Be extremely vigilant when tightening up the resource and data requirements for your MIDlets. This can mean using smaller images, fewer images, or reducing the size of any data that a MIDlet reads from the network or stores for later use.

Optimizing for Speed

Speed optimization is arguably the most important optimization approach for MIDlets because it does the most to determine how well they perform. Speed optimization involves speeding up the execution of MIDlet code by fine-tuning the code. Considering the performance problems inherent in Java, not to mention the relatively slow processors employed by mobile devices, speed optimization plays a critical role in MIDlet development. The Java compiler has the last word on how executable Java bytecode is generated, so most speed optimizations must be performed with the compiler in mind.

New Term *speed optimization*—An optimization technique that involves speeding up the execution of MIDlet code by fine-tuning the code.

Much of today's lesson focuses on issues of speed optimization and how to get the best performance from your MIDlet code. At times, you will sacrifice the other areas of optimization for the sake of speed. In most cases, this sacrifice is entirely acceptable, even expected, because the organization of the code and size of the executable classes won't matter much if a MIDlet is too slow to use. However, you still must strike a balance between optimizing for speed and optimizing for size. A fast MIDlet that is too big to fit within the memory constraints of a mobile device clearly isn't beneficial. Fortunately, size and speed optimizations sometimes go hand-in-hand because simpler algorithms can often be faster and smaller than complex algorithms.

MIDlet Optimization Tips

Before getting into the specifics of how to tweak Java code to squeeze every ounce of performance from it, it is worth taking a moment to cover some general MIDlet optimization strategies that you should consider as you begin designing and developing MIDlets of your own. These strategies focus more on size optimization because the majority of size optimizations take place through the careful design of efficient MIDlets, as opposed to tricky algorithmic code changes. Don't worry, there is plenty of coding trickery in store for you later in the lesson!

Simplify Your MIDlets

The best optimization you can consider making is to simplify your MIDlets—creating MIDlets with simpler user interfaces and simplified information requirements. Try to reduce the number of components and commands used throughout a MIDlet, as well as tighten up the data that the MIDlet reads and writes. Keep simplification at the forefront during MIDlet design by remembering that you are programming phones and pagers; they are devices with small screens and extremely limited "keyboards." It only makes sense that MIDlets should be extremely simple compared to their desktop and handheld computer counterparts.

The best time to focus on MIDlet simplification is before you write a line of code. Study the manner in which the MIDlet is likely to be used, keeping in mind that someone out there will be driving down the road trying to run your MIDlet on their phone. Knowing this, take the time to think through the user interfaces for your MIDlets and streamline them as much as possible. If for no other reason, do it in the name of traffic safety! This suggestion definitely has more than a hint of reality behind it.

Minimize MIDlet Data

You are no doubt aware that MIDP devices currently rely on wireless networking connections that provide relatively slow transfer speeds. Because MIDlets must communicate over this connection, serious limitations exist as to how much data can be sent and received without causing delays. The last thing a user of a MIDlet wants is to have to wait for their device to download time-critical information such as severe weather warnings. The solution is to minimize the size of data used by a MIDlet. This might sound obvious, but most Web sites aren't necessarily designed around the concept of minimizing the size of data.

Most MIDlets don't need to concern themselves with using images as MIDlet data. A good example is the Directions MIDlet that provides directions to people while they are driving. This MIDlet relies on text driving directions, which take up considerably less space than a map image. There is certainly an argument that a map image would be more intuitive in this MIDlet, but that depends largely on the size of the device screen, not to mention whether it is color. More important to this discussion is the fact that a map image would be considerably larger than the text directions, and therefore take longer to download. This is a good example of how the data used by a MIDlet was minimized to make the MIDlet more responsive to the user.

Reduce MIDlet Memory Usage

It's no secret that MIDP devices have very little memory as compared to most other computing environments. In many ways, the memory limitations of MIDP devices are more constraining than the limited processing power of the devices. It is therefore extremely important to try to reduce the memory usage of MIDlets whenever possible. Fortunately, you can employ several practical development techniques to reduce the amount of memory required of a MIDlet. These techniques include:

- Avoid using objects whenever possible.
- When you do use objects, try to recycle them.
- Explicitly clean up objects when you're finished with them.

The next few sections explore these MIDlet memory-reduction techniques in more detail.

Avoid Objects Whenever Possible

This might seem like a strange suggestion, but it is a good idea to avoid the use of objects in MIDlets whenever possible. Objects must be allocated from runtime memory, as opposed to primitive data types, which are allocated directly on the stack. Primitive data types include standard Java language types such as `int`, `long`, `boolean`, and `char`. Of course, the CLDC and MIDP APIs are full of classes, and MIDlets themselves are objects, so there are obvious limitations as to how much you can reduce the usage of objects in MIDlets. However, the reduction in object usage has more to do with MIDlet data, which in many cases can be stored in primitive data types as opposed to full-blown objects.

NEW TERM *the stack*—A special area of system memory that is typically more efficient than runtime memory.

If you study the CLDC and MIDP APIs, you will find that many of the familiar classes used as helper classes throughout the J2SE API are missing. For example, the `Rectangle` class is used throughout the J2SE API as a means of housing the four integers (x, y, width, and height) that describe a rectangular shape. This class is missing in the MIDP API, and in places where the `Rectangle` class would have been used you use the four integers directly. The memory requirements of the four primitive integers are less than the memory requirements of an object with memory that must be allocated and managed. Therefore, you will notice throughout the CLDC and MIDP APIs that objects are only used when it clearly makes functional sense to use an object. Otherwise, primitive data types are used.

Note

Primitive data types are sometimes referred to as *scalar data types*, or just *scalars*.

You should follow the lead of the CLDC and MIDP APIs when it comes to your own MIDlet data. Don't jump to encapsulate data in a class unless there is a significant reason to do so. Otherwise, you'll be much better off to stick with primitive data types, which are much more efficient than objects.

When You Do Use Objects, Recycle Them

Clearly there is no way to avoid the use of objects entirely in your MIDlets. Objects play an extremely important role in all Java programming, and MIDlets are no different in this regard. One way to minimize the memory impact of objects is to recycle them whenever possible. Object recycling is reusing an existing object instead of creating a new one. Of course, this only works in situations where you need to use an object of the same type repeatedly, but you would be surprised at how often this technique can be pulled off in a typical MIDlet.

New Term *object recycling*—The process of reusing an existing object instead of creating a new one, which avoids an unnecessary memory allocation.

Object recycling avoids an unnecessary memory allocation. As an example, if you create an object and then quit using it, the Java garbage collector will eventually free the memory allocated for it. If you then need another object of the same type and you create a new one, the memory for the object is allocated all over again. Instead, you could reinitialize the original object instead of creating a new one, thereby recycling the object.

Clean Up After Yourself

Speaking of recycling and garbage collection, one last optimization tip related to objects involves the manner in which objects are freed from memory. In traditional J2SE or J2EE programming you create objects at will and rely on Java's garbage collector to detect when an object is no longer being used, and clean it up. J2ME works the same way, but the garbage collector is not exactly the most efficient means of freeing up memory. The garbage collector runs as a low priority background thread that detects and frees unused objects every so often. An object isn't deemed "unused" until it goes out of scope or is set to `null`.

One way to help the garbage collector is to explicitly set unused objects to `null` so that the garbage collector can go ahead and free the memory for them as soon as possible. All objects eventually get freed from memory, but this little trick helps the garbage collector to clean up objects from memory a little faster.

Java Optimization Techniques

Thus far I've discussed MIDlet optimization in general terms, without giving any concrete examples of how to tweak MIDlet code to get the most out of it. In the next few sections I present some detailed coding techniques you can use to squeeze additional performance out of your MIDlets. Most of these techniques focus on speeding up MIDlet code, as opposed to reducing the size of it, but that's okay because the tweaks will probably be made in isolated places.

This is an important point because you really shouldn't focus on optimizing every bit of MIDlet code for speed. There are probably isolated sections of code that are called much more often than others. Knowing this, you should focus your optimization efforts on the code that is called the most. Keep in mind that it isn't realistic to think that you'll be able to optimize every piece of problem (slow) code in a MIDlet; the goal is to make big dents in the areas that can be optimized without an enormous amount of effort.

Note

Do not worry too much about optimizing any code in a MIDlet for size or speed until you have the MIDlet working. Optimized code can often involve tricky algorithms that are difficult to debug. Therefore, you should always start with working code when it comes time to perform code optimization tricks.

Eliminate Unnecessary Evaluations

The first code optimization technique I'd like to introduce has to do with unnecessary evaluations. These evaluations are problematic because all they do is eat up precious processor time. Following is an example of some code that unnecessarily performs an evaluation that acts effectively as a constant:

```
for (int i = 0; i < size(); i++)
  a = (b + c) / i;
```

The addition of `b + c`, although itself a pretty efficient piece of code, is better off being calculated before the loop, like this:

```
int tmp = b + c;
for (int i = 0; i < size(); i++)
  a = tmp / i;
```

This simple change could have fairly dramatic effects, depending on how many times the loop is iterated. There is one other optimization you might have missed. Notice that `size()` is a method call, which might bring to mind the performance costs involved in calling a method versus simply checking a variable. You might not realize it, but `size()` is being called every time through the loop as part of the conditional loop expression. The same technique used to eliminate the unnecessary addition operation can be used to fix this problem. Check out the resulting code:

```
int s = size();
int tmp = b + c;
for (int i = 0; i < s; i++)
  a = tmp / i;
```

Eliminate Common Subexpressions

While we are on the subject of optimizing expressions, consider another expression problem that robs your code of performance: common subexpressions. You can use a costly subexpression repeatedly in your MIDlet code without realizing the consequences. In the heat of programming, it's easy to reuse common subexpressions instead of storing them in a temporary variable, like this:

```
b = Math.abs(a) * c;
d = e / (Math.abs(a) + b);
```

The multiple calls to `Math.abs()` are costly compared to calling it once and storing the result in a temporary variable, like this:

```
int tmp = Math.abs(a);
b = tmp * c;
d = e / (tmp + b);
```

Take Advantage of Local Variables

You might not realize it, but it takes longer for MIDlet code to access local variables than it does to access member variables. The reason for this deals with how the two different types of variables are stored in memory. The practical implication is that you should use local variables instead of member variables whenever performance is critical. For example, if you have a loop that accesses a member variable repeatedly then you might consider storing the value of the member variable in a local variable just before the start of the loop, and then accessing the local variable within the loop. Following is an example of code that accesses a member variable in a loop:

```
for (int i = 0; i < 1000; i++)
  a = obj.b * i;
```

As you can see, the `b` member variable of the `obj` object is accessed one thousand times within the loop. A quick optimization of this code involves setting the `obj.b` to a local variable, and then accessing that variable in the loop, like this:

```
int localb = obj.b;
for (int i = 0; i < 1000; i++)
  a = localb * i;
```

Expand Loops

A final speed optimization trick is known as *loop expansion*, and involves expanding a loop to get rid of the overhead involved in maintaining the loop. You might wonder exactly what overhead this refers to. Even a simple counting loop has the overhead of performing a comparison and an increment each time through. This might not seem like much, but every little bit of performance can matter in some MIDlets, such as games.

New Term *loop expansion*—The process of expanding a loop to get rid of the inherent overhead involved in maintaining the loop; also known as loop unrolling.

Loop expansion involves replacing a loop with the brute-force equivalent. To better understand it, check out the following piece of code:

```
for (int i = 0; i < 1000; i++)
  a[i] = 25;
```

This probably looks like some pretty efficient code, and in fact it is. But if you want to go the extra distance and perform a loop expansion on it, here's one approach:

```
int i = 0;
for (int j = 0; j < 100; j++) {
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
  a[i++] = 25;
}
```

In this code, you've reduced the loop overhead by an order of magnitude (from 1000 to 100), but you've introduced some new overhead by having to increment the new index variable (`i`) inside the loop. Overall, this code does outperform the original code, but don't expect any miracles. Loop expansion can be effective at times, but I don't recommend placing it too high on your list of optimization tricks. Again, it only applies to the most extreme MIDlets that are scratching and clawing for every millisecond of improved performance.

Putting MIDlet Optimization into Perspective

Now that you have an understanding of MIDlet optimization, not to mention a few techniques for optimizing MIDlet code, it's time to assess the overall significance of MIDlet optimization. To the extent that you design MIDlets that are relatively efficient and not memory hogs, it is very important to consider the optimization strategies discussed earlier in this lesson. However, I don't recommend focusing on the latter speed optimization techniques unless you feel as if your MIDlet is too slow. In other words, it's always good to reduce the size and memory requirements of a MIDlet, but don't unnecessarily complicate the code to improve speed.

These speed optimization tricks are not exactly good programming habits because they add complexity to MIDlet code, and are also more difficult to maintain. Unless your MIDlet is noticeably dragging its feet, it isn't worth the extra programming effort and coding complexity to use most of these techniques. Having said that, there are some MIDlets where you will want to pull out all the stops and perform every optimization trick in the book. These usually are MIDlets that are pushing the envelope of what can be done on a mobile device. Action games are a good example of this kind of MIDlet because they will be taxing the graphics, processor, and memory of a mobile device to its fullest.

One last issue relating to MIDlet optimization is a bit of a confession. If you study the sample code for this book you won't find much in the way of specific code optimizations. The reality is that code optimizations often add complexity, and the focus of this book is to teach you how MIDlet code works. So, please don't think I'm lazy or hypocritical when you find that much of the code in this book isn't optimized. An area where I do focus heavily on optimization is on the overall design of the MIDlets. You'll find all the MIDlets throughout this book to be relatively straightforward and simplistic in their requirements, which is the best MIDlet optimization of all.

Summary

This lesson departed from the heavy coding of the past few lessons by tackling an interesting topic related to MIDlets: optimization. There are several facets to MIDlet optimization that impact both design and coding, but there is no doubt that optimization at some level is an important concern for every MIDlet developer. This lesson began by introducing the basics of optimization as it applies to MIDlets. You then learned some general strategies for optimizing MIDlets primarily so that they are smaller and use less memory. From there you moved on to learn some specific coding techniques that enable you to squeeze every bit of performance out of MIDlet Java code. Finally, the lesson concluded by revisiting the general notion of MIDlet optimization and putting its relevance into perspective.

Part III, "Wireless Information Management with MIDlets," introduces you to persistent storage and how MIDlets can be used to store and retrieve data. You also work through the development of several very practical MIDlets such as a contact manager, a financial check register, and an auction manager.

Q&A

Q Do all MIDlets require lots of code optimization to run at acceptable speeds?

A No. In fact, most MIDlets don't require any speed optimizations at all. Only MIDlets that are really stretching the bounds of what mobile devices can accomplish are targets for intensive speed optimizations. Having said this, virtually all MIDlets can benefit from being optimized to reduce their size and memory requirements.

Q Why can't I optimize code while I'm writing it to begin with?

A Technically speaking, you can. However, it can sometimes be challenging enough to get code debugged and working properly without worrying about optimizations. Attempting to carry out tricky optimizations on code that isn't completely debugged is dangerous and can induce serious headaches. For the record, I had a good friend that would jokingly brag about optimizing code as he wrote it. He was actually able to get away with it most of the time, but he had programming talents that far exceeded mine. For the rest of us mortals, it's best to write it, test it, and then optimize it.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What are the three major areas of code optimization?
2. What is the least important type of optimization for MIDlets?
3. As an optimization technique, how is it possible to avoid using objects in a MIDlet?
4. What is object recycling?

Exercises

1. Choose a MIDlet that's been covered in a previous lesson, and assess it in terms of the three types of optimization. In other words, examine the MIDlet and see how well it is implemented in terms of being optimized for maintainability, size, and speed.

2. Take this same MIDlet and apply a few of the specific MIDlet optimization techniques that you learned about in this lesson. For example, see if you can find any objects to recycle, and maybe a few more that can be set to `null` to help out the Java garbage collector.

PART III

Wireless Information Management with MIDlets

13 Using the MIDP Record Management System (RMS)

14 Staying in Touch with Your Contacts

15 Managing Your Finances

16 Bidding on the Go

DAY 13

Using the MIDP Record Management System (RMS)

One issue that I have skirted throughout the book thus far is persistent data storage—how to store information in a MIDlet so that you can come back to it later. Although the concept of persistent storage is anything but new, the very nature of mobile wireless devices prevents you from making assumptions about how data is stored. For example, persistent storage on a desktop or laptop computer is handled with hard disks, floppy disks, and writeable CDs, none of which are available for most mobile devices. What to do?

This lesson reveals the solution to the persistent storage problem for MIDP devices by introducing you to the MIDP Record Management System (RMS). The RMS is a simple database system that enables you to store and retrieve information to a special database file that is stored in the memory of a device. The RMS is both powerful and easy to use at the same time, which makes it possible to incorporate it into your MIDlets without too much trouble. The following major topics will be covered in this lesson as you get acquainted with the MIDP RMS:

- Understanding the MIDP Record Management System (RMS)
- Exploring the classes and interfaces in the RMS package
- Storing data persistently with record stores
- Designing and constructing the ToDoList MIDlet

What Is the MIDP Record Management System (RMS)?

The MIDP Record Management System (RMS) is a set of classes and interfaces that provide support for a simple database system that is geared to storing MIDlet data. MIDlets are relatively small applications in the grand scheme of things, so their data needs aren't expected to be overly complicated. Most mobile devices do not have enough resources to allow for much complication. Even so, it is still important for the MIDP API to provide MIDlets with a means of storing information *persistently* so that the information can be retrieved later. The RMS is the portion of the MIDP API that makes this possible.

NEW TERM *persistent storage*—The process of storing information in a non-volatile location so that it can later retrieved.

The main premise behind the RMS and its persistent storage features is to allow MIDlet data to exist after a MIDlet is terminated. You don't want your data to go away just because you close a MIDlet or turn your phone off. Persistent MIDlet data is stored in such a way that it is kept around after the MIDlet that created it is exited. When the MIDlet is later run again, the data magically appears again. Actually, there is no magic behind persistent storage and the RMS. In fact, it works basically the same as a database on a desktop or laptop computer, except that in the case of a MIDP device the database file is technically stored somewhere in the device's memory, as opposed to being stored in a file on a hard drive.

You might wonder why the J2ME architects didn't just use the JDBC database features that are built into J2SE and J2EE. JDBC is extremely powerful and has many features for storing and retrieving persistent data, so surely it is capable of doing anything a MIDlet could need to have done. JDBC is a quite capable technology, but its powerful database features actually become a hindrance when you look at what it takes to carry out those features. Not only is JDBC a relatively large API, but it also requires a fair amount of processing power to operate properly. Part of this is because JDBC includes support for

SQL databases, which require a considerable amount of overhead, and are beyond the needs of MIDlets. Given the extreme constraints of MIDP devices, it didn't make sense to use such a comprehensive database API when all MIDlets need is a simple means of storing data persistently. And that's how the RMS was born.

The fundamental unit of storage in the RMS is a *record*, which is stored in a special database called a *record store*. A record is actually an array of bytes, whereas a record store is a binary file containing a collection of records. When you create a MIDlet that uses RMS for persistent storage, you first create a record store, and then you add individual records to it. These records are then safely stored away so that you can retrieve them from the record store later.

NEW TERM *record*—A unit of data that is stored in a RMS database (record store).

NEW TERM *record store*—A RMS database that is used to store records.

When a record is added to a record store it is assigned a unique number that is referred to as a *record ID*. Record IDs are important because they provide a means of efficiently identifying records in a record store. Many of the methods in the RMS API that are used to interact with record stores rely on record IDs to manage individual records in a record store.

NEW TERM *record ID*—A number that is used to uniquely identify a record in a record store.

Inside the RMS Package

The MIDP API includes a package devoted solely to the RMS—`javax.microedition.rms`. This package includes a class and several interfaces that provide the framework for records, record stores, and other RMS features. More specifically, the `javax.microedition.rms` package provides the following primary persistent storage capabilities for MIDlets:

- Enables MIDlets to add records to and remove records from a record store
- Enables MIDlets in the same MIDlet suite to access each other's record stores

The one class defined in the `javax.microedition.rms` package is `RecordStore`, which (not surprisingly) represents a record store. In addition to this class, several interfaces are included in the package that carry out various RMS-related functions such as enumerating and filtering records. The next few sections examine the `RecordStore` class and the RMS interfaces in more detail.

The `RecordStore` Class

The `RecordStore` class represents a record store, which is actually a binary file that is stored in the memory of a MIDP device. The `RecordStore` class enables you to open, close, and delete record stores. Additionally, you can use the `RecordStore` class to add, get, and delete records, as well as to enumerate the records in the record store. All these tasks are made possible through methods in the `RecordStore` class. The following methods represent the most commonly used `RecordStore` methods, which we will put to use throughout the remainder of the lesson:

- **`openRecordStore()`**—Opens a record store
- **`closeRecordStore()`**—Closes the record store
- **`deleteRecordStore()`**—Deletes the record store
- **`getName()`**—Obtains the name of the record store
- **`getNumRecords()`**—Obtains the number of records in the record store
- **`addRecord()`**—Adds a record to the record store
- **`getRecord()`**—Gets a record from the record store
- **`deleteRecord()`**—Deletes a record in the record store
- **`enumerateRecords()`**—Obtains an enumeration for the record store

Although other `RecordStore` methods exist, you will find that these methods are the ones that come into play the most often when working with record stores. A little later in the lesson in the section titled "Working with Record Stores," you learn how to put these methods to use.

The RMS Interfaces

In addition to the `RecordStore` class, the `javax.microedition.rms` package has several useful interfaces, including the following:

- **`RecordEnumeration`**—Describes a means of enumerating through a record store
- **`RecordComparator`**—Describes a means of comparing two records
- **`RecordFilter`**—Describes a filter that can be used to sift records from a record store
- **`RecordListener`**—An event listener that receives event notifications whenever a record is added, changed, or deleted from a record store

Each of these interfaces plays an important role in the RMS system, but the most significant one is `RecordEnumeration`, which enables you to traverse a record store as an enumeration. When you step through a record store as an enumeration, you basically move through the records one at a time. You can move forward or backward through the record store, as well as obtain the ID or data for each record. The following methods are defined in the `RecordEnumeration` interface:

- **`hasNextElement`**`()`—Returns `true` if another element is in the following direction
- **`hasPreviousElement`**`()`—Returns `true` if another element is in the previous direction
- **`nextRecordId`**`()`—Obtains the next record ID
- **`nextRecord`**`()`—Obtains the next record data
- **`previousRecordId`**`()`—Obtains the previous record ID
- **`previousRecord`**`()`—Obtains the previous record data
- **`reset`**`()`—Resets the enumeration to the beginning

The next section, "Working with Record Stores," explains how these methods are used to traverse a record store as an enumeration.

Working with Record Stores

Like most databases, the principal operations performed on a record store include opening and closing the record store, as well as adding, getting, and deleting records. Additionally, it can be very useful to traverse through the records in a record store. The next few sections explore the specific code required to carry out these different record store tasks.

Opening a Record Store

The first step in any MIDlet that uses the RMS is to open a record store by calling the static `openRecordStore()` method in the `RecordStore` class. In addition to opening existing record stores, the `openRecordStore()` method enables you to create a new record store and open it. The following is an example of how to open a new record store named `lotsadata`:

```
RecordStore recordStore = RecordStore.openRecordStore("lotsdata", true);
```

The first parameter to the `openRecordStore()` method is a string specifying the name of the record store, and the second parameter is a Boolean value specifying that the record store should be created if it doesn't already exist. The `openRecordStore()` method returns an object of type `RecordStore`, which can then be used to work with the record store. The name of a record store must be 32 characters or less, and is case sensitive. This name forms the basis for the binary database file, which is ultimately how the record store is stored persistently. The name of the database file is the record store name with the file extension `.db`. As an example, the `lotsadata` record store is stored in device memory under the name `lotsadata.db`.

Note

When using the J2ME Wireless Toolkit to emulate RMS MIDlets, the record store database files are stored in the directory named `nojam` that appears beneath the main J2ME Wireless Toolkit directory. As an example, the `lotsadata.db` file would be stored in the `\j2mewtk\nojam` directory if you installed the J2ME Wireless Toolkit to the default installation directory.

Adding New Records

To add a record to a record store, you must first have the record data in the proper format, which is a byte array. Adding new records to a record store is a matter of calling the `addRecord()` method on the `RecordStore` object and passing in the byte array for the record. The following code snippet illustrates how to add a record to a record store using the `addRecord()` method:

```
int id = 0;
try {
  id = recordStore.addRecord(bytes, 0, bytes.length);
}
catch (RecordStoreException e) {
  e.printStackTrace();
}
```

The `addRecord()` method requires you to pass in the byte array as the first parameter, the offset into the array as the second parameter, and the number of bytes to add as the last parameter. To add all the bytes in an array, you pass in `0` as the offset and `bytes.length` as the number of bytes to add. The return value of the `addRecord()` method is the ID of the newly added record, which uniquely identifies the record in the record store.

Note

The record ID for a record store serves as the primary key for the database. The IDs begin at 1 for the first record added, and then increase by 1 for each record added after that.

Keep in mind that a record must be stored as an array of bytes before it can be added to a record store. In some cases you might create a record from string data, in which case you must convert the `String` object to a byte array before adding it to a record store.

Fortunately, the `String` class includes a method named `getBytes()` that performs this exact task. Following is an example of converting a string named `str` to a byte array:

```
byte[] bytes = str.getBytes();
```

Getting Records

Storing records in a record store wouldn't be of much use if you couldn't retrieve them from the record store later. Getting a record from a record store involves a call to the `getRecord()` method, which obtains a record given a record ID. The actual information returned from the `getRecord()` method is a byte array containing the record data. The following code snippet shows how to use the `getRecord()` method:

```
byte[] recordData = null;
try {
  recordData = recordStore.getRecord(id);
}
catch (RecordStoreException e) {
  e.printStackTrace();
}
```

This code assumes that you know the ID of the record that you would like to get. Typically, a MIDlet will maintain a list of record IDs that it builds when it first reads through the record store. This isn't strictly necessary but it helps to keep a record store synchronized to a GUI component such as a list. It is also possible to search through a record store to find a record, in which case you obtain its ID that way. Both approaches require you to enumerate the record store, which you learn how to do shortly.

Deleting Records

Similar to getting a record, deleting a record requires you to know the ID of the record you would like to delete. With the ID in hand, you can delete the record with a call to the `deleteRecord()` method. This code snippet gives an example of how to delete a record by calling the `deleteRecord()` method:

```
try {
  recordStore.deleteRecord(id);
}
catch (RecordStoreException e) {
  e.printStackTrace();
}
```

When a record is deleted, it is completely removed from the record store. The RMS does not support advanced database features such as transactions and rollbacks, so be careful when you delete records because there is no way to resurrect a deleted record.

Enumerating Records

It is often necessary to enumerate a record store so that you can step through its records. This is useful if you want to fill a list with records, or if you want to search through a record store for a certain record. The `enumerateRecords()` method in the `RecordStore` class makes it possible to enumerate a record store. This method returns an object that implements the `RecordEnumeration` interface, which is what you use to interact with the enumerated records. The following example shows how you can obtain an enumerated set of records from a record store through the `enumerateRecords()` method:

```
RecordEnumeration records = recordStore.enumerateRecords(null, null, false);
```

The three parameters to the `enumerateRecords()` method are used to customize the enumeration process. More specifically, the first parameter is a `RecordFilter` object that is used to filter the records included in the enumeration. The second parameter is a `RecordComparator` object that determines the order in which the records are enumerated; this enables you to sort the records if desired. The last parameter of `enumerateRecords()` is a Boolean value that determines whether the enumeration is kept updated with the record store. If you set this parameter to `true`, the enumeration will be automatically updated to reflect changes in the record store such as the addition and deletion of records. If you plan to keep an enumeration around for long, it is a good idea to use a value of `true` for this parameter so that the enumerated records are kept in sync with the record store. However, if you are going to quickly enumerate a record store and then let go of the enumeration, it is better to use a value of `false` so that there is less overhead for the RMS.

Note

Using `true` for the third parameter of the `enumerateRecords()` method forces the RMS to keep the enumeration synchronized with the record store, which requires a considerable amount of processing. For this reason, you should only use this approach if you know there is a risk of the record store being modified while you are still using the enumeration. If you are simply enumerating a record store and then letting go of the enumeration, it is better performance-wise not to force the synchronization issue.

The previous code example shows how to call the `enumerateRecords()` method to obtain an enumeration that contains all the records in a record store in the order in which they were added. To step through the records in the enumeration, you must use several methods defined in the `RecordEnumeration` interface. For example, the `hasNextElement()` method checks to see whether there is another record to traverse, which is important as you step through the records. You step through the records by calling either `nextRecord()` or `nextRecordId()`. Both of these methods start with the first record in the enumeration the first time they are called and move to the next record with each subsequent call; the difference between the two is that `nextRecord()` returns a byte array containing the current record data, whereas `nextRecordId()` returns the current record ID.

The following code snippet demonstrates how to use a record enumeration to step through the records in a record store and print out all the record IDs:

```
RecordEnumeration records = null;
try {
  records = recordStore.enumerateRecords(null, null, false);
  while(records.hasNextElement())
    int id = records.nextRecordId();
    System.out.println(id);
}
catch(Exception e) {
  System.err.println("EXCEPTION: Problem reading the task records.");
}
```

Although this code only prints the ID of each record in the record store, it nonetheless lays the groundwork for how to enumerate a record store and get information about each record. You could easily modify this code to search for a particular record ID and obtain the record data for the record when found.

Closing the Store

It is important to close a record store when you are finished using it. The `RecordStore` class provides a method named `closeRecordStore()` that carries out this task. The `closeRecordStore()` method takes no parameters and doesn't return anything, so its use is quite simple:

```
recordStore.closeRecordStore();
```

That is all it takes to close a record store. You now know enough about working with records and record stores to work through a practical example.

The ToDoList Sample MIDlet

You can probably imagine storing and accessing many things in a mobile device on a regular basis. As you now know, anything that you would like to have around beyond the lifecycle of a given MIDlet must be stored persistently using the MIDP RMS. A neat example of something that fits in perfectly with the role of a mobile device is a simple to-do list, which is basically a list of tasks that you need to accomplish. This to-do list could contain tasks as mundane as errands you need to run or as important as business meetings scheduled for the week. In this section you work through the design and development of a ToDoList MIDlet that uses the RMS to persistently store a to-do list.

The key to the ToDoList MIDlet is keeping it simple enough so that it is both quick and easy to add and manage tasks. Considering that mobile devices have relatively small screens, it doesn't make sense to include very much information for a given task. In fact, it is probably sufficient to simply include a due date for the task along with text that describes the task. The task text should also be very concise because mobile devices aren't exactly easy to type on. A task record could therefore consist of a date and a piece of text describing the task. The main screen of the MIDlet could be a list that shows all the tasks, along with their due dates. Keep in mind that the main premise of the ToDoList MIDlet is to show how to use the RMS to store and retrieve MIDlet data persistently.

Managing the Task Records

The heart of the ToDoList MIDlet is the record store that stores the tasks for the to-do list. It is therefore very important to spend some time making sure the task data is formatted and stored properly. To help organize the MIDlet better, it is helpful to encapsulate the task information in a class of its own. Additionally, it also makes the MIDlet code cleaner to create a wrapper class that makes interacting with the record store a little easier. These two classes are as follows:

- **`ToDoTask`**—Represents a single task in the to-do list
- **`ToDoDB`**—Encapsulates the record store that stores task records

The following sections describe the details of these two classes, including the source code that brings them to life.

The `ToDoTask` Class

The `ToDoTask` class represents a task in the to-do list, and is responsible for formatting the task data so that it can be stored to and retrieved from the record store with little effort. Listing 13.1 represents the code for the `ToDoTask` class.

LISTING 13.1 The `ToDoTask` Class Represents a Task in the To-Do List

```
import java.util.*;

public class ToDoTask {
  private Date date;
  private String task;
```

LISTING 13.1 continued

```
    public ToDoTask(Date d, String t) {
      date = d;
      task = t;
    }

    public ToDoTask(byte[] data) {
      unpack(new String(data));
    }

    public void unpack(String data) {
      int index = data.indexOf(';');
      date = new Date(Long.parseLong(data.substring(0, index)));
      task = data.substring(index + 1, data.length());
    }

    public String pack() {
      return (String.valueOf(date.getTime()) + ';' + task);
    }

    public String getFormattedTask() {
      // Convert the date member to a Calendar object
      Calendar c = Calendar.getInstance();
      c.setTime(date);

      // Return the formatted task
      String s = (c.get(Calendar.MONTH) + 1) + "/" + c.get(Calendar.DAY_OF_MONTH)
  +
        " " + task;
      return s;
    }
  }
```

The two member variables of the `ToDoTask` class, `date` and `task`, are relatively straightforward in that they store the date and text for the task. These variables are initialized in the first constructor for the `ToDoTask` class, which accepts a `Date` object and a `String` object as parameters. The second `ToDoTask()` constructor is a little more interesting in that it accepts a byte array as its only parameter. As you might guess, this constructor enables you to create a `ToDoTask` object from a task record that has been read from a record store. To extract the task data from the byte array, this constructor converts the byte array to a string and then calls the `unpack()` method.

The `unpack()` method extracts task data from a record string. To understand how this method works, you must first understand how task data is packed into a record string to begin with. This is accomplished by the `pack()` method, which creates a string out of the `date` and `task` member variables. The only tricky thing about this method is the date of the task, which is stored as a `Date` object. To store this value, you must obtain it as a numeric date. This is done by calling the `getTime()` method on the `date` member variable, which returns a date as a `long` integer. The string representation of the task is then created as a numeric date followed by a semicolon followed by the task text. So, a record string returned by the `pack()` method looks something like this:

```
12345678;CHANGE OIL
```

Now that you know how a task is packed into a string, you can better understand how the `unpack()` method works. The `unpack()` method parses a record string and extracts the numeric date and task text. The numeric date is then converted into a `Date` object so that it can be stored in the date member variable.

The last method in the `ToDoTask` class is the `getFormattedTask()` method, which is responsible for formatting a task for display purposes. This method can be used by a MIDlet to obtain a task string that is suitable for displaying the task in a list. The formatted task string consists of a month and day followed by the task text. The following is an example of a formatted task:

```
7/15 CHANGE OIL
```

Although a formatted task is admittedly somewhat terse, you must consider that there isn't much room on mobile device displays. That wraps up the `ToDoTask` class. Let's move on to see how the `ToDoDB` class makes it easier to work with a task-based record store.

The `ToDoDB` Class

The `ToDoDB` class is a means of managing a task-based record store. Although there's nothing stopping you from directly working with a `RecordStore` object at the MIDlet level, a helper class like `ToDoDB` can significantly simplify MIDlet code by hiding many of the details of interacting with the record store. The `ToDoDB` class takes on several responsibilities when it comes to managing the task record store. More specifically, it enables you to carry out the following record store operations:

- Open the record store
- Close the record store
- Add a task record
- Delete a task record
- Obtain an enumeration of the record store

Listing 13.2 supplies the code for the `ToDoDB` class, which reveals how each of these operations are carried out.

LISTING 13.2 The ToDoDB Class Manages a Task-based Record Store

```
import javax.microedition.rms.*;
import java.util.Enumeration;
import java.util.Vector;
import java.io.*;

public class ToDoDB {
  RecordStore recordStore = null;

  public ToDoDB(String name) {
    // Open the record store using the specified name
    try {
      recordStore = open(name);
    }
    catch(RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public RecordStore open(String fileName) throws RecordStoreException {
    return RecordStore.openRecordStore(fileName, true);
  }

  public void close() throws RecordStoreNotOpenException,
    RecordStoreException {
    // If the record store is empty, delete the file
    if (recordStore.getNumRecords() == 0) {
      String fileName = recordStore.getName();
      recordStore.closeRecordStore();
      recordStore.deleteRecordStore(fileName);
    }
    else {
      // Otherwise, close the record store
      recordStore.closeRecordStore();
    }
  }

  public synchronized int addTaskRecord(String record) {
    // Convert the string record to an array of bytes
    byte[] bytes = record.getBytes();

    // Add the byte array to the record store
    try {
      return recordStore.addRecord(bytes, 0, bytes.length);
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }
```

LISTING 13.2 continued

```
      return -1;
    }

    public synchronized void deleteTaskRecord(int id) {
      // Delete the task record from the record store
      try {
        recordStore.deleteRecord(id);
      }
      catch (RecordStoreException e) {
        e.printStackTrace();
      }
    }

    public synchronized RecordEnumeration enumerateTaskRecords()
      throws RecordStoreNotOpenException {
      return recordStore.enumerateRecords(null, null, false);
    }
  }
```

The `ToDoDB` class contains a single member variable, `recordStore`, which is the record store for the ToDoList MIDlet. The `ToDoDB()` constructor takes on the role of opening the record store by calling the `open()` method. Notice that the name of the record store database is passed into the constructor and used as the basis for opening the record store. The `open()` method carries out the details of opening the record store, which basically means calling the static `openRecordStore()` method in the `RecordStore` class. The second parameter of the `openRecordStore()` method is passed as `true` so that a new database file will be created if none exists.

The `ToDoDB` class provides a `close()` method that enables you to close the record store. The `close()` method is probably not as simplistic as you might have expected, but that's because it carries out an extra function. The `close()` method checks to see whether any records are in the record store before it closes it. If no records are in the record store then there is no reason to keep the database file around, so the `close()` method deletes it with a call to `deleteRecordStore()`. If records are in the record store then the `close()` method closes it.

The `addTaskRecord()` method is the next method of interest in the `ToDoDB` class. It is not too much of a leap to guess that this method adds a task record to the record store, which is exactly what it does. The method accepts the task record as a string, which you know must be converted to a byte array to be added to the record store. So, the first step in the `addTaskRecord()` method is to convert the task string to a byte array. The byte array is then added to the record store through a call to the `addRecord()` method. It is very important to notice the return value of the addTaskRecord() method, which is the ID of the newly added record. This ID is important because it will be used in the ToDoList MIDlet to maintain a list of record IDs for the record store.

The next ToDoDB method on the agenda is deleteTaskRecord(), which deletes a task record from the record store based on a supplied record ID. This record ID is passed as the only parameter to the deleteTaskRecord() method.

The last method in the ToDoDB class is called enumerateTaskRecords(), and is used to obtain an enumeration of the task records. The enumerateRecords() method is called on the record store to obtain an enumeration of all the records in the record store.

Note

You might have noticed that the addTaskRecord(), deleteTaskRecord(), and enumerateTaskRecords() methods are all defined as being synchronized methods. This doesn't really impact the ToDoList MIDlet, but it would be important in a MIDlet that used threads to interact with a record store.

The User Interface

I have alluded to the fact that the ToDoList MIDlet must include a list of tasks as its main screen, which should give you a pretty good clue about the structure of the MIDlet's user interface. In addition to the main screen's task list, the MIDlet also needs a form that enables you to fill out the details of a task (date and text). This form should be displayed when adding a new task. These user interface elements are declared as member variables of the ToDoList class, along with a few other important variables:

```
private Command exitCommand, addCommand, backCommand, saveCommand;
private Display display;
private List mainScreen;
private Form taskScreen;
private DateField dateField;
private TextField taskField;
private ToDoDB db = null;
private Vector taskIDs = new Vector();
```

As you can see, the MIDlet includes several commands that enable you to interact with tasks. The Exit and Add commands appear on the main screen and the Back and Save commands appear on the task form screen. Selecting the Add command displays the task form screen, where the user can enter a new task. The new task can then be added to the record by selecting the Save command, or cancelled by using the Back command. In addition to these commands, the Display object for the MIDlet is declared, along with the `List` object for the main screen and the `Form` object for the task screen. The `DateField` and `TextField` variables are used to establish the user interface for the task screen.

The `ToDoDB` member variable is the record store itself, whereas the last member variable, `taskIDs`, is a vector that performs an interesting role with respect to the user interface. Each task record in the record store appears as an item in the list on the main screen of the MIDlet. Although you know the ID of a record when you retrieve it from the record store, there is no way to associate the record ID with its respective item in the list. However, the only way to delete a task is to let the user select it from the list, and then delete its record from the record store. This requires that you know the ID of the task in the list. The solution to this problem is to maintain a list of task IDs that parallels the tasks in the list. Then, when the user selects an item from the list to delete, you can look up its record ID in the `taskIDs` vector and delete it from the record store.

Now that you understand the member variables in the MIDlet, you can move on to the `ToDoList()` constructor, which puts the member variables into perspective (see Listing 13.3).

LISTING 13.3 The `ToDoList()` Constructor Initializes the ToDoList MIDlet

```
public ToDoList() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  addCommand = new Command("Add", Command.SCREEN, 3);
  backCommand = new Command("Back", Command.BACK, 3);
  saveCommand = new Command("Save", Command.OK, 3);

  // Create the main screen
  mainScreen = new List("To Do List", List.IMPLICIT);

  // Set the Exit and Add commands for the main screen
  mainScreen.addCommand(exitCommand);
  mainScreen.addCommand(addCommand);
  mainScreen.setCommandListener(this);

  // Create the task screen
  taskScreen = new Form("New Task");
  dateField = new DateField("", DateField.DATE);
  taskScreen.append(dateField);
  taskField = new TextField("", "", 20, TextField.ANY);
  taskScreen.append(taskField);
```

LISTING 13.3 continued

```
    // Set the Back and Save commands for the task screen
    taskScreen.addCommand(backCommand);
    taskScreen.addCommand(saveCommand);
    taskScreen.setCommandListener(this);

    // Open the ToDo database
    try {
      db = new ToDoDB("tasks");
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem opening the database.");
    }

    // Read through the database and build a list of record IDs
    RecordEnumeration records = null;
    try {
      records = db.enumerateTaskRecords();
      while(records.hasNextElement())
        taskIDs.addElement(new Integer(records.nextRecordId()));
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the task records.");
    }

    // Read through the database and fill the task list
    records.reset();
    try {
      while(records.hasNextElement()) {
        ToDoTask task = new ToDoTask(records.nextRecord());
        mainScreen.append(task.getFormattedTask(), null);
      }
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the task records.");
    }
  }
```

As you can see, this is a pretty hefty constructor. It is tackling many initialization tasks for the MIDlet, however, so its size is justified. The first step in the `ToDoList()` constructor is to get the `Display` object for the MIDlet. The commands are then created with appropriate priority levels; the `Exit` command is given a higher priority than the others. From there, the main screen is created, followed by the task screen, and commands are added to each. With the screens and commands set up properly, the `ToDoList()` constructor shifts gears a little by building a list of record IDs. As you learned earlier, this list is necessary so that you can ascertain the record ID of items in the task list. The code that builds the list of record IDs is similar to the record enumeration example you saw earlier in the lesson.

After enumerating the record store and building a vector of record IDs, the `ToDoList()` constructor resets the enumeration and steps through the records one more time. This second enumeration is necessary to add the formatted tasks to the task list. Notice that a new `ToDoTask` object is created for each record in the record store, after which the task is formatted and added to the main task list.

Handling Commands

All the commands in the ToDoList MIDlet are handled in the `commandAction()` method. The `Exit` command takes on the job of closing down the MIDlet, whereas the `Add` command enables the user to enter information for a new task. The `Back` and `Save` commands apply only to the task form and are used to either cancel adding a new task or add the new task to the record store. One other command of interest is the `Select` command, which isn't a command defined by the MIDlet but is instead generated when you select an item in the task list by clicking the Select button on a mobile device. To make it easy to quickly remove tasks, the `Select` command in the ToDoList MIDlet is used to delete the currently selected task in the task list. Listing 13.4 shows the `commandAction()` method, which illustrates how each of these command features is implemented.

LISTING 13.4 The `commandAction()` Method Handles Commands for the ToDoList MIDlet

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == addCommand) {
    // Initialize the task fields
    dateField.setDate(Calendar.getInstance().getTime());
    taskField.setString("");

    // Set the current display to the task screen
    display.setCurrent(taskScreen);
  }
  else if (c == List.SELECT_COMMAND) {
    // Get the record ID of the currently selected task
    int index = mainScreen.getSelectedIndex();
    int id = ((Integer)taskIDs.elementAt(index)).intValue();

    // Delete the task record
    db.deleteTaskRecord(id);
    taskIDs.removeElementAt(index);
    mainScreen.delete(index);
  }
  else if (c == backCommand) {
    // Set the current display back to the main screen
```

LISTING 13.4 continued

```
        display.setCurrent(mainScreen);
      }
      else if (c == saveCommand) {
        // Create a record for the task and save it to the database
        ToDoTask task = new ToDoTask(dateField.getDate(), taskField.getString());
        taskIDs.addElement(new Integer(db.addTaskRecord(task.pack())));
        mainScreen.append(task.getFormattedTask(), null);

        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
    }
```

This is another hefty method. However, you will be glad to know that this method houses the majority of the functionality in the ToDoList MIDlet. The code for the `Exit` command is pretty familiar to you by now, so I won't elaborate. The `Add` command code is a little more interesting even though it doesn't involve interacting with the record store. This is because you need to provide the user with a means of canceling the new record. So, the record isn't actually added to the record store until you select the `Save` command from the task form screen. The `Add` command code simply initializes the task form for the new task by setting the date field to the current date and clearing the task field. It then sets the task form screen as the current screen so that it is displayed.

The Select command code is where the `commandAction()` method starts doing some interesting things with the record store. When the user issues this command, a task in the task list has been selected for deletion. The Select command code must first obtain the currently selected task from the list and then look up its record ID in the `taskIDs` vector. After that's done, all you have to do is pass this record ID into the `deleteTaskRecord()` method to delete the task record from the record store. Of course, it is also necessary to remove the record ID for the task from the `tasksIDs` vector, as well as delete the task from the main task list.

The Back command code is used to cancel the addition of a task record and return to the main task list screen. This results in a single line of code that calls the `setCurrent()` method on the `Display` object.

The last command handled in the `commandAction()` method is the `Save` command, which takes the task entered into the task form and adds it to the record store. The code for this command begins by creating a `ToDoTask` object based on the information entered into the date and task fields of the task form. The task is then added to the record store, and its record ID is added to the `taskIDs` vector. The new task is then formatted and added to the main task list. Finally, the Save command code finishes by setting the current display back to the main screen.

Cleaning Up

At this point you've seen virtually all the code for the ToDoList MIDlet. However, some cleanup work remains to be done. If you recall from earlier in the lesson, it is important to close a record store when you're finished using it. The record store in the ToDoList MIDlet must be closed when the MIDlet exits. Therefore, it makes sense to add the code that closes the record store to the `destroyApp()` method, which follows in Listing 13.5.

LISTING 13.5 The `destroyApp()` Method Takes Care of Closing the Record Store when the MIDlet Exits

```
public void destroyApp(boolean unconditional) {
  // Close the ToDo database
  try {
    db.close();
  }
  catch(Exception e) {
    System.err.println("EXCEPTION: Problem closing the database.");
  }
}
```

The `destroyApp()` method closes the record store with a single call to the `close()` method of the `ToDoDB` object.

Putting It All Together

Although you have seen the most important parts of the code for the ToDoList MIDlet, you haven't seen the MIDlet code in its entirety. For the sake of understanding how everything fits together, it's worth taking a quick look at the complete source code for the ToDoList MIDlet, which follows in Listing 13.6.

LISTING 13.6 The Complete Source Code for the ToDoList MIDlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;

public class ToDoList extends MIDlet implements CommandListener {
  private Command exitCommand, addCommand, backCommand, saveCommand;
  private Display display;
  private List mainScreen;
  private Form taskScreen;
```

LISTING 13.6 continued

```
    private DateField dateField;
    private TextField taskField;
    private ToDoDB db = null;
    private Vector taskIDs = new Vector();

    public ToDoList() {
      // Get the Display object for the MIDlet
      display = Display.getDisplay(this);

      // Create the commands
      exitCommand = new Command("Exit", Command.EXIT, 2);
      addCommand = new Command("Add", Command.SCREEN, 3);
      backCommand = new Command("Back", Command.BACK, 3);
      saveCommand = new Command("Save", Command.OK, 3);

      // Create the main screen
      mainScreen = new List("To Do List", List.IMPLICIT);

      // Set the Exit and Add commands for the main screen
      mainScreen.addCommand(exitCommand);
      mainScreen.addCommand(addCommand);
      mainScreen.setCommandListener(this);

      // Create the task screen
      taskScreen = new Form("New Task");
      dateField = new DateField("", DateField.DATE);
      taskScreen.append(dateField);
      taskField = new TextField("", "", 20, TextField.ANY);
      taskScreen.append(taskField);

      // Set the Back and Save commands for the task screen
      taskScreen.addCommand(backCommand);
      taskScreen.addCommand(saveCommand);
      taskScreen.setCommandListener(this);

      // Open the ToDo database
      try {
        db = new ToDoDB("tasks");
      }
      catch(Exception e) {
        System.err.println("EXCEPTION: Problem opening the database.");
      }

      // Read through the database and build a list of record IDs
      RecordEnumeration records = null;
      try {
        records = db.enumerateTaskRecords();
        while(records.hasNextElement())
          taskIDs.addElement(new Integer(records.nextRecordId()));
```

LISTING 13.6 continued

```
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the task records.");
    }

    // Read through the database and fill the task list
    records.reset();
    try {
      while(records.hasNextElement()) {
        ToDoTask task = new ToDoTask(records.nextRecord());
        mainScreen.append(task.getFormattedTask(), null);
      }
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the task records.");
    }
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the main screen
    display.setCurrent(mainScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
    // Close the ToDo database
    try {
      db.close();
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem closing the database.");
    }
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == addCommand) {
      // Initialize the task fields
      dateField.setDate(Calendar.getInstance().getTime());
      taskField.setString("");

      // Set the current display to the task screen
      display.setCurrent(taskScreen);
```

LISTING 13.6 continued

```
      }
      else if (c == List.SELECT_COMMAND) {
        // Get the record ID of the currently selected task
        int index = mainScreen.getSelectedIndex();
        int id = ((Integer)taskIDs.elementAt(index)).intValue();

        // Delete the task record
        db.deleteTaskRecord(id);
        taskIDs.removeElementAt(index);
        mainScreen.delete(index);
      }
      else if (c == backCommand) {
        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
      else if (c == saveCommand) {
        // Create a record for the task and save it to the database
        ToDoTask task = new ToDoTask(dateField.getDate(), taskField.getString());
        taskIDs.addElement(new Integer(db.addTaskRecord(task.pack())));
        mainScreen.append(task.getFormattedTask(), null);

        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
    }
  }
```

The bulk of the MIDlet's code is still contained within the `ToDoList()` constructor and the `commandAction()` method, which isn't too surprising now that you understand what they accomplish. Keep in mind that this code is still dependent on the `ToDoTask` and `ToDoDB` classes that you saw earlier in the lesson. Those two classes combined with the `ToDoList` MIDlet class complete the ToDoList MIDlet. That can only mean one thing—you're ready to see the MIDlet in action in the J2ME emulator.

After compiling, pre-verifying, and packaging the ToDoList MIDlet, you're ready to try it out in the J2ME emulator. Figure 13.1 shows the ToDoList MIDlet after first running it in the emulator.

As the figure shows, when you first run the ToDoList MIDlet the task list is empty because you have yet to add any tasks. To add a task, select the `Add` command. The New Task screen (task form) is then displayed, as shown in Figure 13.2.

The date field in the New Task screen is initialized with the current date. To change the date for the task, click the Select button on the device. The visual editor for the date field is displayed, and is shown in Figure 13.3.

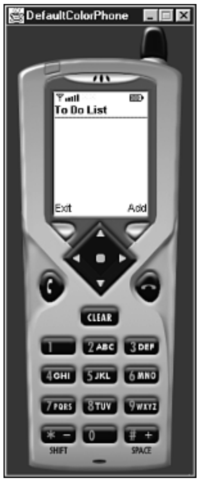


FIGURE 13.1

When you run it for the first time, the task list in the ToDoList MIDlet is empty.

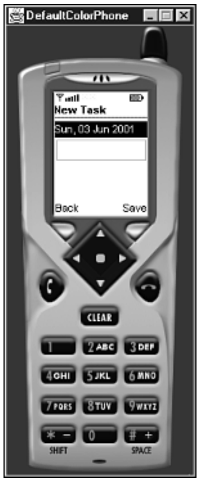


FIGURE 13.2

The New Task screen includes a date field and a text field for entering the task.

After selecting a date, you'll be returned to the New Task screen, where you can move on to entering text for the task. Figure 13.4 shows the New Task screen after entering text for the task.

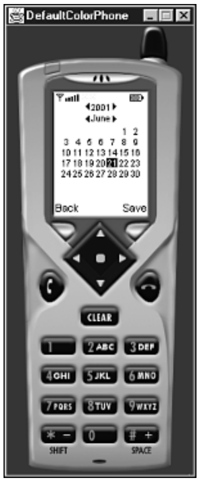


FIGURE 13.3
The visual editor for the date field lets you select a date using a graphical calendar interface.

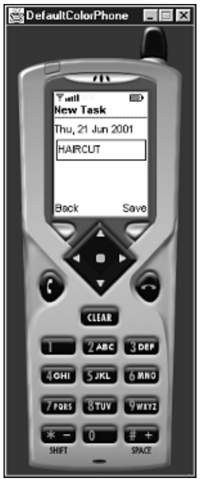
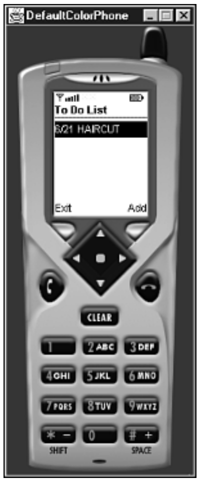


FIGURE 13.4
After selecting a date, you must enter the text for the task in the New Task screen.

After the date and text for the task have been entered, you can select the `Save` command to add the task to the record store, or use the `Back` command to cancel out of the task addition. Figure 13.5 shows the main task list after selecting the `Save` command to add the new task to the record store.

FIGURE 13.5

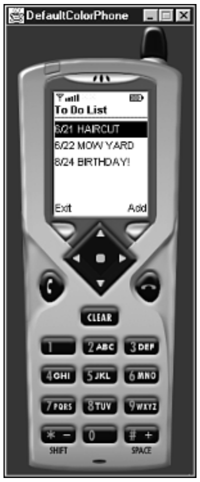
Adding a new task to the record store involves selecting the Add command from the New Task screen, after which the task will appear in the main task list.



If you continue to enter additional tasks, they will appear in the task list one after the next. Figure 13.6 shows the task list after entering a couple of more tasks.

FIGURE 13.6

As you add more tasks, they are each added to the task list on the main screen.



If you recall, the Select button lets you delete tasks from the record store (and the task list). To delete a task, use the arrow keys to navigate to the desired task and then click the Select button on the device. Figure 13.7 shows the main task list after deleting the "MOW YARD" task.

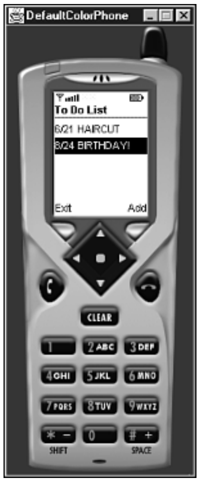


FIGURE 13.7
When you delete a task from the record store, it is also removed from the task list.

That sums up the functionality of the ToDoList MIDlet, which is a good example of how to store data persistently. If you want to check and make sure that the to-do list data has really been stored away safely, try closing the MIDlet and restarting it. You will find that the tasks appear in the task list as if you never quit the MIDlet. You now have the power to create MIDlets that can remember things!

Summary

I am sure you know that memory and data storage are incredibly important parts of any computer system. All the work that is input must somehow be saved so that when the computer is turned off it doesn't go away. Although mobile devices typically aren't the first things that come to mind when thinking of productivity applications, there is nonetheless information that it would be nice to store on such devices. Therefore, it is important for MIDlets to have some means of storing information so that it doesn't go away when a MIDlet exits or when the device is powered off. This type of storage technique is known as persistent storage, and is the basis for the MIDP Record Management System (RMS).

This lesson introduced you to the RMS and what it has to offer in terms of adding persistent storage features to MIDlets. You learned about the different classes and interfaces that go into the RMS API, as well as how to use them to manage record stores, which are the MIDP equivalent of databases. The lesson concluded by guiding you through the design and development of a to-do list MIDlet that enables you to store a list of tasks. Your newfound RMS knowledge will come in handy in the next lesson as you construct a MIDlet to manage a list of contacts.

Q&A

Q Can record stores be shared across multiple MIDlets?

A Yes. However, only MIDlets within the same MIDlet suite can share a record store. In other words, MIDlets from different MIDlet suites cannot access each other's record stores.

Q How do you ensure that a record store isn't lost when a mobile device's battery runs down?

A Unfortunately, you don't really have any direct control over how a device responds to losing power. It is the responsibility of the MIDP implementation on the device to ensure that no persistent data is lost when a device loses power, but ultimately there is no guarantee that this won't happen. The safest thing to do is make sure the device never goes completely without power!

Q In the ToDoList MIDlet, is it essential that the task data be stored as a string?

A No. Keep in mind that records in a record set are ultimately just byte arrays, which means that you could feasibly maintain record data as raw bytes. I used strings in the ToDoList MIDlet so that it would be easier to see how the data is converted back and forth from a record store to data fields of the `ToDoTask` class.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What is the fundamental unit of storage in the MIDP Record Management System (RMS)?
2. What is the significance of record IDs?
3. What MIDP package houses the classes and interfaces for the RMS API?
4. In what form are records stored in a record store?

Exercises

1. Modify the ToDoList MIDlet so that tasks can be flagged as completed. Hint: This involves adding another data member to the ToDoTask class, as well as displaying a special image (checkmark?) next to each completed task in the main task list.
2. Modify the ToDoList MIDlet so that it sorts the tasks according to the due date. Hint: This involves using a `RecordComparator` when enumerating the task records.

DAY 14

Staying in Touch with Your Contacts

Most mobile phones enable you to enter a list of phone numbers that you commonly use, along with the name of the person associated with the number. Although this primitive contact list is useful for making calls to familiar friends and business associates, it isn't powerful enough to serve as a general contact directory. For one thing, there is no way to enter additional numbers, such as a fax number, for a person. Also, there is no way to organize the numbers according to the relationship you have with the person. For example, I have my contact list organized so that personal, family, and business contacts are grouped separately. As you might guess, it's possible to develop a handy contact manager as a MIDlet—that's the premise of this lesson.

This lesson guides you through the design and development of a MIDlet called Contacts that serves as a contact manager for mobile devices. If you use a personal digital assistant such as a Pocket PC or Palm device, you are no doubt already familiar with carrying your contacts around electronically. The Contacts MIDlet takes the concept of mobile contacts a step further by enabling you to manage and store these contacts on your mobile phone or pager. The following primary topics are covered in this lesson:

- Assessing the need for a mobile contact manager
- Designing and coding the Contacts MIDlet
- Managing your contacts with the Contacts MIDlet

Managing Contacts on the Go

Even those of us who aren't seasoned business travelers can appreciate the significance of having the phone numbers and related contact information for friends, family, and business associates within close reach at all times. My mobile phone enables me to enter several phone numbers for people that I call regularly, but there isn't much flexibility when it comes to entering additional information such fax numbers, not to mention the difficulty of managing the contact list. Because mobile phones and pagers haven't been the ideal contact managers, many people use handheld computers such as Pocket PCs and Palm devices to manage their contacts. Other people, such as my wife, are a little more traditional in that they rely on entering contact information by hand into travel planners.

NEW TERM *contact manager*—An application that is used to keep track of a list of contacts, typically including the phone numbers, mailing address, and e-mail address of each contact.

No single approach to managing contacts is better than another approach. It is a personal preference, but using your mobile phone as a contact manager is ideal in many ways because you are more likely to have your phone handy than you are to have a handheld computer or travel planner handy. A phone is what you actually use to contact people in your contact list, so it makes sense to have the information stored on the phone. The only significant drawback to storing contact information on a mobile phone is that entering the information is somewhat tedious because of the limited "keyboard" on such devices. However, entering contact information is more or less a one-time chore for each person in the list.

If you have ever used a full-featured contact manager such as Microsoft Outlook, you might be thinking that there is much information potentially being stored with each contact on a mobile device. However, a mobile contact list isn't intended to catalog every piece of information potentially associated with a contact. For example, I use Microsoft Outlook to store the birthdays of friends and family members, but this doesn't exactly qualify as the kind of information that I want to have accessible from a mobile contact list. So, a mobile contact list definitely needs to store scaled-down contact information, just as most MIDlets are significantly scaled down relative to their desktop-equivalent applications.

Knowing that a contact manager MIDlet should store a minimal amount of contact information, what do you consider minimal information for a contact? Your idea of "minimal" might differ from mine, but the following is a list of minimal contact information that I'd like to have available in a mobile contact list:

- Name
- Company
- Phone number
- Fax number
- Contact type

You could definitely argue that the company for a contact is not an essential requirement in a mobile contact list, but traveling sales people with many contacts might appreciate this information. Also, the type of the contact might not seem all that important to you, but it might be helpful in enabling you to sort or filter contacts in the contact list according to type. If you only wanted to view business contacts, the contact type would provide a means of doing so. One thing that you might think is missing in this list is a mailing address. I was tempted to include it, but a mobile contact list should provide a means of contacting someone while you're on the go, which typically isn't through the mail. Nonetheless, adding a mailing address to the contact information wouldn't necessarily be a bad idea.

Now that you understand the significance of having a contact manager MIDlet, and the kind of information it should store for each contact, let's start assembling the Contacts MIDlet.

Constructing the Contacts MIDlet

As you might be thinking, the Contacts MIDlet is similar in many ways to the ToDoList MIDlet from the previous lesson. Both MIDlets must store and retrieve information persistently, and both must display the information in a list on the main screen of the MIDlet. Knowing these similarities, the first thing to understand about the Contacts MIDlet is that it will use the RMS to store contact information persistently. Beyond that, the user interface for the Contacts MIDlet is somewhat more advanced than the ToDoList MIDlet because more information is associated with a contact than with a task in a to-do list.

Another important distinction between the Contacts and ToDoList MIDlets is that you cannot edit tasks in the ToDoList MIDlet. These two MIDlets are so simple that it makes more sense to just delete a task and create a new one if you want to change it. However, it is very important to allow the user to edit a contact because it is fairly common for contact information to change as people move and change jobs. So, the user interface for the Contacts MIDlet must somehow support editing contacts, in addition to adding and deleting them.

Managing the Contact Records

It shouldn't come as too much of a surprise that the Contacts MIDlet relies heavily on a couple of classes to represent an individual contact and a database of contacts. This is the same approach used in the ToDoList MIDlet, and it makes sense to use it here as well. More specifically, the following two classes are used by the Contacts MIDlet to make it easier for the MIDlet to manage and interact with contact records:

- **`Contact`**—Represents a single contact in the contact list
- **`ContactDB`**—Encapsulates the record store that stores contact records

These classes are explained in detail in the next two sections. The code will look somewhat familiar as it roughly parallels the code for the `ToDoTask` and `ToDoDB` classes that you learned about in the previous lesson.

The `Contact` Class

It is helpful to use a class to represent each contact in the contact list so that you can easily access the contact information from the Contacts MIDlet. The `Contact` class handles this task by storing the information associated with a contact and providing several handy methods for accessing and manipulating this information. The methods in the `Contact` class enable you to access the different pieces of contact information individually, as well as pack and unpack contact data for persistent storage and retrieval in a record store. The code for the `Contact` class is in Listing 14.1.

LISTING 14.1 The `Contact` Class Represents a Contact in the Contact List

```
import java.util.*;

public class Contact {
  private String name, company, phone, fax;
  private int type;

  public Contact(String n, String c, String p, String f, int t) {
    name = n;
    company = c;
```

LISTING 14.1 continued

```
    phone = p;
    fax = f;
    type = t;
  }

  public Contact(byte[] data) {
    unpack(new String(data));
  }

  public void unpack(String data) {
    int start = 0, end = data.indexOf(';');
    name = data.substring(start, end);
    start = end + 1;
    end = data.indexOf(';', start);
    company = data.substring(start, end);
    start = end + 1;
    end = data.indexOf(';', start);
    phone = data.substring(start, end);
    start = end + 1;
    end = data.indexOf(';', start);
    fax = data.substring(start, end);
    start = end + 1;
    type = Integer.parseInt(data.substring(start, data.length()));
  }

  public String pack() {
    return (name + ';' + company + ';' + phone + ';' + fax + ';' +
      ((String)Integer.toString(type)));
  }

  public String getName() {
    return name;
  }

  public String getCompany() {
    return company;
  }

  public String getPhone() {
    return phone;
  }

  public String getFax() {
    return fax;
  }

  public int getType() {
    return type;
  }
}
```

The relevant information for a contact is stored in the member variables of the Contact class: `name`, `company`, `phone`, `fax`, and `type`. The `type` member variable is an integer instead of a string because you will be use predefined contact types (Friend, Family, Business, and so forth) to set its value. The member variables for the `Contact` class are initialized in the first `Contact()` constructor, through which you pass each piece of information as a parameter. The second constructor is much more interesting because it initializes a `Contact` object using a byte array of contact data. This data is in fact a contact record that is read from a record store, which means the constructor you create a `Contact` object directly from a record. The specifics of the data extraction are left up to the `unpack()` method, which operates on a string containing contact information.

Before learning about the `unpack()` method, let's first take a look at how contact records are stored. When a contact record is stored in a record store, it is first formatted into a string using the `pack()` method. The `pack()` method basically places the contact information in a string with each piece of information separated by a semicolon. To visualize this, look at the following line, which is an example of how a contact string might appear:

```
MILTON JAMES;CAFE COCO;321-2626;555-2626;1
```

You can see in this example how each piece of contact information is simply listed in the string, separated by semicolons. Now that you understand how a contact record is packed into a string for storage, you can work backward to figure out how the `unpack()` method extracts data from a packed string. The `unpack()` method parses a contact record string and extracts the name, company, phone, fax, and type data. The numeric type is handled a little differently from the others because it must be converted from a string to a number; this is carried out by the `Integer.parseInt()` method.

The remaining methods in the `Contact` class are access methods that enable you to retrieve the different pieces of contact information. These methods come in handy to display a contact in a contact list. You learn how this works later in the lesson when you get into the details of the `Contacts` MIDlet class.

Now that you have a feel for how the `Contact` class models an individual contact, let's continue and find out how the `ContactDB` class provides a means of managing a contact record store.

The `ContactDB` Class

You learned in the previous lesson that things are a little cleaner for a MIDlet if a helper class hides some details of working with a record store. The `ContactDB` class plays the role of record store helper in the Contacts MIDlet by managing the contact record store. In keeping with the role of the `ToDoDB` class in the ToDoList MIDlet from the previous lesson, following are the record store tasks taken care of by the `ContactDB` class:

- Open the record store
- Close the record store
- Retrieve a contact record
- Add a contact record
- Delete a contact record
- Obtain an enumeration of the record store

The code for the `ContactDB` class in Listing 14.2 reveals how each of these tasks is accomplished.

LISTING 14.2 The `ContactDB` Class Manages a Record Store of Contacts

```
import javax.microedition.rms.*;
import java.util.Enumeration;
import java.util.Vector;
import java.io.*;

public class ContactDB {
  RecordStore recordStore = null;

  public ContactDB(String name) {
    // Open the record store using the specified name
    try {
      recordStore = open(name);
    }
    catch(RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public RecordStore open(String fileName) throws RecordStoreException {
    return RecordStore.openRecordStore(fileName, true);
  }

  public void close() throws RecordStoreNotOpenException,
    RecordStoreException {
    // If the record store is empty, delete the file
    if (recordStore.getNumRecords() == 0) {
      String fileName = recordStore.getName();
      recordStore.closeRecordStore();
      recordStore.deleteRecordStore(fileName);
    }
    else {
      // Otherwise, close the record store
      recordStore.closeRecordStore();
```

LISTING 14.2 continued

```
    }
  }

  public Contact getContactRecord(int id) {
    // Get the contact record from the record store
    try {
      return (new Contact(recordStore.getRecord(id)));
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }

    return null;
  }

  public synchronized void setContactRecord(int id, String record) {
    // Convert the string record to an array of bytes
    byte[] bytes = record.getBytes();

    // Set the record in the record store
    try {
      recordStore.setRecord(id, bytes, 0, bytes.length);
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public synchronized int addContactRecord(String record) {
    // Convert the string record to an array of bytes
    byte[] bytes = record.getBytes();

    // Add the byte array to the record store
    try {
      return recordStore.addRecord(bytes, 0, bytes.length);
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }

    return -1;
  }

  public synchronized void deleteContactRecord(int id) {
    // Delete the contact record from the record store
    try {
      recordStore.deleteRecord(id);
    }
```

LISTING 14.2 continued

```
    catch (RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public synchronized RecordEnumeration enumerateContactRecords()
    throws RecordStoreNotOpenException {
    return recordStore.enumerateRecords(null, null, false);
  }
```

The only member variable in the `ContactDB` class, `recordStore`, is used to store the record store for the Contacts MIDlet. The `ContactDB()` constructor initializes this member variable by opening the record store based on the name that is passed in the `name` parameter. The record store is opened with a call to the `open()` method, which in turn calls the static `openRecordStore()` method in the `RecordStore` class.

It is important to close a record store after you are finished with it. The `ContactDB` class enables you to close the contact record store with the `close()` method. Before closing the record store, the `close()` method first checks to see whether any records are in it. If no records are in the record store, the `close()` method first deletes the record store by calling `deleteRecordStore()`. If records are in the record store, the `close()` method closes it with a call to `closeRecordStore()`.

The `getContactRecord()` method retrieves a contact record from the record store. The method accepts a record ID as its only parameter, and then uses this ID to get the record from the record store and create a `Contact` object based on the record data.

The `setContactRecord()` method alters an existing contact record in the record store by setting it with new contact data. The method accepts the ID and new record string for the contact record. This record string is converted to a byte array and then used to set the record.

The `addContactRecord()` method adds a new contact record to the record store. The method accepts the new contact record as a string, which is then converted to a byte array before being added to the record store. The `addContactRecord()` method returns the record ID of the newly added record, which is important for keeping track of records after adding them to a record store.

Just as contacts can be added with the `addContactRecord()` method, they can also be deleted using the `deleteContactRecord()` method. This method deletes a contact record from the record store based upon a record ID that is passed as the only parameter to the method.

The only remaining method in the ContactDB class is the enumerateContactRecords() method, which is used to obtain an enumeration of the contact records. This method calls the enumerateRecords() method on the record store to obtain an enumeration of all the contact records in the record store.

The User Interface

With the contact helper classes taken care of, you can now move into the details of the Contacts MIDlet class. The first aspect of the MIDlet class to tackle is its user interface code, which establishes the MIDlet's screens and input fields. You already know that the MIDlet will consist of a main screen that lists all the contacts in the record store. It is also necessary to provide a means of editing and adding new contacts, and this involves an additional screen with input fields for each of the pieces of contact information. This contact screen should be implemented as a Form object because it must contain several other GUI components (input fields).

The following code snippet lists the member variables for the Contacts MIDlet, which provide some insight into the user interface of the MIDlet:

```
private Command exitCommand, addCommand, backCommand, saveCommand,
  deleteCommand;
private Display display;
private List mainScreen;
private Form contactScreen;
private TextField nameField, companyField, phoneField, faxField;
private ChoiceGroup contactType;
private boolean editing = false;
private ContactDB db = null;
private Vector contactIDs = new Vector();
private Image[] typeImages;
```

The first few variables represent the commands for the MIDlet, which enable you to interact with contacts and exit the MIDlet. The Exit and Add commands appear on the main screen, while the Back, Save, and Delete commands appear on the contact form screen. Selecting the Add command results in the contact form screen being displayed, where the user can enter a new contact. The new contact can then be added to the record by selecting the Save command, or cancelled by using the Back command. To edit a contact, use the Select button on the device, which issues the standard Select command. When this takes place, the Delete command is added to the contact screen so that the user can delete the contact if so desired.

In addition to the commands, the Display object for the MIDlet is declared as a member variable, along with the List object for the main screen and the Form object for the contact screen. Several text fields are then created for the various pieces of contact information, along with a choice group for the contact type. This choice group is initialized with standard values in the MIDlet constructor in just a moment. The `editing` member variable is used by the MIDlet to determine whether the user is adding or editing a contact. This is necessary because the contact screen is used for both purposes.

The `ContactDB` member variable is declared next, along with `contactIDs`, which is a vector that keeps track of the record ID for each contact in the contact list. If you recall from the ToDoList MIDlet, this variable is necessary because there is no way to associate a contact in the list with a record in the record store without having a record ID. The last member variable in the `Contacts` MIDlet is `typeImages`, which stores an array of images that are used to provide a graphical icon for each contact in the list. The icon indicates the type of a contact, which means that the `typeImages` array contains images that correspond to each contact type.

The `Contacts()` constructor handles the details of initializing the member variables and getting the MIDlet set up. The code for the `Contacts()` constructor is in Listing 14.3.

LISTING 14.3 The `Contacts()` Constructor Initializes the Contacts MIDlet

```
public Contacts() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  addCommand = new Command("Add", Command.SCREEN, 3);
  backCommand = new Command("Back", Command.BACK, 2);
  saveCommand = new Command("Save", Command.OK, 3);
  deleteCommand = new Command("Delete", Command.SCREEN, 3);

  // Create the main screen
  mainScreen = new List("Contacts", List.IMPLICIT);

  // Set the Exit and Add commands for the main screen
  mainScreen.addCommand(exitCommand);
  mainScreen.addCommand(addCommand);
  mainScreen.setCommandListener(this);

  // Create the contact screen
  contactScreen = new Form("Contact Info");
  nameField = new TextField("Name", "", 30, TextField.ANY);
  contactScreen.append(nameField);
  companyField = new TextField("Company", "", 15, TextField.ANY);
  contactScreen.append(companyField);
  phoneField = new TextField("Phone", "", 10, TextField.PHONENUMBER);
```

LISTING 14.3 continued

```
    contactScreen.append(phoneField);
    faxField = new TextField("Fax", "", 10, TextField.PHONENUMBER);
    contactScreen.append(faxField);
    String[] choices = { "Personal", "Business", "Family", "Other" };
    contactType = new ChoiceGroup("Type", Choice.EXCLUSIVE, choices, null);
    contactScreen.append(contactType);

    // Set the Back, Save, and Delete commands for the contact screen
    contactScreen.addCommand(backCommand);
    contactScreen.addCommand(saveCommand);
    contactScreen.addCommand(deleteCommand);
    contactScreen.setCommandListener(this);

    // Load the type images
    try {
      typeImages = new Image[4];
      typeImages[0] = Image.createImage("/Personal.png");
      typeImages[1] = Image.createImage("/Business.png");
      typeImages[2] = Image.createImage("/Family.png");
      typeImages[3] = Image.createImage("/Other.png");
    }
    catch (IOException e) {
      System.err.println("EXCEPTION: Failed loading images!");
    }

    // Open the contact database
    try {
      db = new ContactDB("contacts");
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem opening the database.");
    }

    // Read through the database and build a list of record IDs
    RecordEnumeration records = null;
    try {
      records = db.enumerateContactRecords();
      while(records.hasNextElement())
        contactIDs.addElement(new Integer(records.nextRecordId()));
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the contact records.");
    }

    // Read through the database and fill the contact list
    records.reset();
    try {
      while(records.hasNextElement()) {
```

LISTING 14.3 continued

```
        Contact contact = new Contact(records.nextRecord());
        mainScreen.append(contact.getName(), typeImages[contact.getType()]);
      }
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the contact records.");
    }
  }
```

The `Contacts()` constructor does quite a lot of setup work for the MIDlet, as the code reveals. The first step in the `Contacts()` constructor is to obtain the `Display` object for the MIDlet. The commands are then created with appropriate priority levels; the `Exit` and `Back` commands are given a higher priority than the others are so that they will be immediately accessible on the main screen and the contact screen, respectively. The main screen is then created, followed by the contact screen, and commands are added to each. Pay close attention to the creation of the contact screen because it reveals the nature of contact types. The contact types are created as an array of strings—the actual value of a contact type is its index in this string array. The choice group for the contact type is initialized with this string array. The image icons associated with the contact types are loaded just after creating the choice group.

The next task for the `Contacts()` constructor is to open the record store database and build a vector of contact record IDs. As I mentioned earlier, this vector is used to associate a record ID with each contact in the main contact list.

After building the vector of record IDs, the `Contacts()` constructor resets the record store enumeration and steps through the records again. This time around, the enumeration is used to add the contacts to the contact list. Notice that a new `Contact` object is created for each record in the record store, after which the `getName()` and `getType()` methods are used to format the contact before adding it to the main contact list. This is where a small image icon is used to represent the type of a contact in the contact list.

Handling Commands

The `commandAction()` method in the Contacts MIDlet takes on the role of handling all of the MIDlet's commands. The majority of the functionality of the MIDlet is contained within this method. As you know, the `Exit` command is responsible for shutting down the MIDlet, whereas the `Add` command enables the user to enter information for a new contact. The `Delete` command enables you to delete a contact from the record store, but it is only available when you are editing an existing contact. The `Save` command is available both when you add a new contact and when you edit an existing contact, and results in the new or changed contact information being committed to the record store. The `Back` command is used to return to the main screen from the contact screen without saving any changes. The last command handled in the `commandAction()` method is the special `Select` command, which is invoked when the user clicks the Select button on the device. This command is used to edit the currently selected contact.

The code for the `commandAction()` method shows how all of the MIDlet commands are handled (see Listing 14.4).

LISTING 14.4 The `commandAction()` Method Handles Commands for the Contacts MIDlet

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == addCommand) {
    // Clear the contact fields
    nameField.setString("");
    companyField.setString("");
    phoneField.setString("");
    faxField.setString("");
    contactType.setSelectedIndex(0, true);

    // Remove the Delete command from the contact screen
    contactScreen.removeCommand(deleteCommand);
    editing = false;

    // Set the current display to the contact screen
    display.setCurrent(contactScreen);
  }
  else if (c == List.SELECT_COMMAND) {
    // Get the record ID of the currently selected contact
    int index = mainScreen.getSelectedIndex();
    int id = ((Integer)contactIDs.elementAt(index)).intValue();

    // Retrieve the contact record from the database
    Contact contact = db.getContactRecord(id);

    // Initialize the contact fields
    nameField.setString(contact.getName());
    companyField.setString(contact.getCompany());
    phoneField.setString(contact.getPhone());
    faxField.setString(contact.getFax());
    contactType.setSelectedIndex(contact.getType(), true);

    // Add the Delete command to the contact screen
    contactScreen.addCommand(deleteCommand);
    editing = true;
```

LISTING 14.4 continued

```
      // Set the current display to the contact screen
      display.setCurrent(contactScreen);
    }
    else if (c == deleteCommand) {
      // Get the record ID of the currently selected contact
      int index = mainScreen.getSelectedIndex();
      int id = ((Integer)contactIDs.elementAt(index)).intValue();

      // Delete the contact record
      db.deleteContactRecord(id);
      contactIDs.removeElementAt(index);
      mainScreen.delete(index);

      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
    else if (c == backCommand) {
      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
    else if (c == saveCommand) {
      if (editing) {
        // Get the record ID of the currently selected contact
        int index = mainScreen.getSelectedIndex();
        int id = ((Integer)contactIDs.elementAt(index)).intValue();

        // Create a record for the contact and set it in the database
        Contact contact = new Contact(nameField.getString(),
➥companyField.getString(),
          phoneField.getString(), faxField.getString(),
➥contactType.getSelectedIndex());
        db.setContactRecord(id, contact.pack());
        mainScreen.set(index, contact.getName(), typeImages[contact.getType()]);
      }
      else {
        // Create a record for the contact and add it to the database
        Contact contact = new Contact(nameField.getString(),
➥companyField.getString(),
          phoneField.getString(), faxField.getString(),
➥contactType.getSelectedIndex());
        contactIDs.addElement(new Integer(db.addContactRecord(contact.pack())));
        mainScreen.append(contact.getName(), typeImages[contact.getType()]);
      }

      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
  }
```

The first command handled in this relatively large method is the `Exit` command, which exits the MIDlet. The `Add` command is up next, which clears the input fields in the contact screen and sets the current display to the contact screen. Also, the `Delete` command is removed from the contact screen because it doesn't make sense to delete a contact while you're adding it. The `editing` flag is then set to `false` to indicate that the user is adding a contact.

The `Select` command is tackled next, and is interesting because it uses the currently selected contact in the main contact list as the basis for initializing the input fields in the contact screen. The reason for this is that the command enables you to edit a contact, which means the contact data must be read from the record store and displayed in the contact screen. To do this, the ID of the currently selected contact is obtained and used to read a record from the record store, which is then used to create a `Contact` object. This object is then used to set the input fields of the contact screen to the contact information. Also, the `Delete` command is added to the contact screen so that the user has the ability to delete the contact, and the `editing` flag is set to `true`.

Speaking of deleting contacts, the `Delete` command code deletes the currently selected contact from the record store by passing its record ID into the `deleteContactRecord()` method. The record ID for the contact is then removed from the `contactIDs` vector, and the contact is deleted from the main contact list.

The code for the `Back` command cancels the addition or editing of a contact record and returns to the main contact list screen. This is accomplished in a single line of code that calls the `setCurrent()` method on the `Display` object.

The `Save` command is the final command for the Contacts MIDlet, and therefore the last command handled in the `commandAction()` method. The `Save` command either adds a new contact or saves an edited contact to the record store. The code for this command begins by looking to see whether the contact was edited or if it is being added. If the contact was edited, the code for the `Save` command creates a new `Contact` object and uses it to change the contact record in the record set. If the contact is being added, a new `Contact` object is created and used to add a new record to the record set. The `Save` command code finishes by setting the main screen as the current display.

Putting It All Together

As you just saw, the majority of the code in the `Contacts` MIDlet class is contained in the `Contacts()` constructor and the `commandAction()` method. Even so, that doesn't mean that it isn't helpful to see these methods in the context of the entire code for the class. Listing 14.5 is the complete source code for the `Contacts` class.

LISTING 14.5 The Complete Source Code for the `Contacts` Class MIDlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;

public class Contacts extends MIDlet implements CommandListener {
  private Command exitCommand, addCommand, backCommand, saveCommand,
    deleteCommand;
  private Display display;
  private List mainScreen;
  private Form contactScreen;
  private TextField nameField, companyField, phoneField, faxField;
  private ChoiceGroup contactType;
  private boolean editing = false;
  private ContactDB db = null;
  private Vector contactIDs = new Vector();
  private Image[] typeImages;

  public Contacts() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    addCommand = new Command("Add", Command.SCREEN, 3);
    backCommand = new Command("Back", Command.BACK, 2);
    saveCommand = new Command("Save", Command.OK, 3);
    deleteCommand = new Command("Delete", Command.SCREEN, 3);

    // Create the main screen
    mainScreen = new List("Contacts", List.IMPLICIT);

    // Set the Exit and Add commands for the main screen
    mainScreen.addCommand(exitCommand);
    mainScreen.addCommand(addCommand);
    mainScreen.setCommandListener(this);

    // Create the contact screen
    contactScreen = new Form("Contact Info");
    nameField = new TextField("Name", "", 30, TextField.ANY);
    contactScreen.append(nameField);
    companyField = new TextField("Company", "", 15, TextField.ANY);
    contactScreen.append(companyField);
    phoneField = new TextField("Phone", "", 10, TextField.PHONENUMBER);
    contactScreen.append(phoneField);
```

LISTING 14.5 continued

```
    faxField = new TextField("Fax", "", 10, TextField.PHONENUMBER);
    contactScreen.append(faxField);
    String[] choices = { "Personal", "Business", "Family", "Other" };
    contactType = new ChoiceGroup("Type", Choice.EXCLUSIVE, choices, null);
    contactScreen.append(contactType);

    // Set the Back, Save, and Delete commands for the contact screen
    contactScreen.addCommand(backCommand);
    contactScreen.addCommand(saveCommand);
    contactScreen.addCommand(deleteCommand);
    contactScreen.setCommandListener(this);

    // Load the type images
    try {
      typeImages = new Image[4];
      typeImages[0] = Image.createImage("/Personal.png");
      typeImages[1] = Image.createImage("/Business.png");
      typeImages[2] = Image.createImage("/Family.png");
      typeImages[3] = Image.createImage("/Other.png");
    }
    catch (IOException e) {
      System.err.println("EXCEPTION: Failed loading images!");
    }

    // Open the contact database
    try {
      db = new ContactDB("contacts");
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem opening the database.");
    }

    // Read through the database and build a list of record IDs
    RecordEnumeration records = null;
    try {
      records = db.enumerateContactRecords();
      while(records.hasNextElement())
        contactIDs.addElement(new Integer(records.nextRecordId()));
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the contact records.");
    }

    // Read through the database and fill the contact list
    records.reset();
    try {
      while(records.hasNextElement()) {
        Contact contact = new Contact(records.nextRecord());
```

LISTING 14.5 continued

```
        mainScreen.append(contact.getName(), typeImages[contact.getType()]);
      }
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the contact records.");
    }
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the main screen
    display.setCurrent(mainScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
    // Close the contact database
    try {
      db.close();
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem closing the database.");
    }
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == addCommand) {
      // Clear the contact fields
      nameField.setString("");
      companyField.setString("");
      phoneField.setString("");
      faxField.setString("");
      contactType.setSelectedIndex(0, true);

      // Remove the Delete command from the contact screen
      contactScreen.removeCommand(deleteCommand);
      editing = false;

      // Set the current display to the contact screen
      display.setCurrent(contactScreen);
    }
    else if (c == List.SELECT_COMMAND) {
      // Get the record ID of the currently selected contact
      int index = mainScreen.getSelectedIndex();
      int id = ((Integer)contactIDs.elementAt(index)).intValue();
```

Listing 14.5 continued

```
        // Retrieve the contact record from the database
        Contact contact = db.getContactRecord(id);

        // Initialize the contact fields
        nameField.setString(contact.getName());
        companyField.setString(contact.getCompany());
        phoneField.setString(contact.getPhone());
        faxField.setString(contact.getFax());
        contactType.setSelectedIndex(contact.getType(), true);

        // Add the Delete command to the contact screen
        contactScreen.addCommand(deleteCommand);
        editing = true;

        // Set the current display to the contact screen
        display.setCurrent(contactScreen);
      }
      else if (c == deleteCommand) {
        // Get the record ID of the currently selected contact
        int index = mainScreen.getSelectedIndex();
        int id = ((Integer)contactIDs.elementAt(index)).intValue();

        // Delete the contact record
        db.deleteContactRecord(id);
        contactIDs.removeElementAt(index);
        mainScreen.delete(index);

        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
      else if (c == backCommand) {
        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
      else if (c == saveCommand) {
        if (editing) {
          // Get the record ID of the currently selected contact
          int index = mainScreen.getSelectedIndex();
          int id = ((Integer)contactIDs.elementAt(index)).intValue();

          // Create a record for the contact and set it in the database
          Contact contact = new Contact(nameField.getString(),
➥companyField.getString(),
            phoneField.getString(), faxField.getString(),
➥contactType.getSelectedIndex());
          db.setContactRecord(id, contact.pack());
          mainScreen.set(index, contact.getName(), typeImages[contact.getType()]);
        }
```

LISTING 14.5 continued

```
      else {
        // Create a record for the contact and add it to the database
        Contact contact = new Contact(nameField.getString(),
➥companyField.getString(),
          phoneField.getString(), faxField.getString(),
➥contactType.getSelectedIndex());
        contactIDs.addElement(new Integer(db.addContactRecord(contact.pack())));
        mainScreen.append(contact.getName(), typeImages[contact.getType()]);
      }

      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
  }
}
```

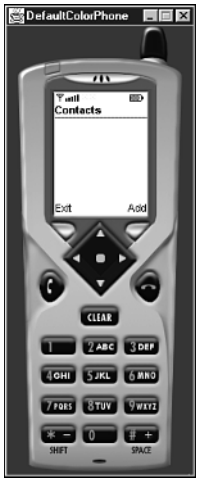
When added to the `Contact` and `ContactDB` classes, this code represents the entirety of the Contacts MIDlet. Although it is a fair amount of code, you must consider that the Contacts MIDlet is quite useful as a contact manager for MIDlet devices. To determine its usefulness for you, test it out in the J2ME emulator.

Testing the Contacts MIDlet

The Contacts MIDlet is one of the most practical MIDlets you've seen throughout the book thus far, and therefore is an interesting MIDlet to test in the J2ME emulator. After you've compiled, pre-verified, and packaged the MIDlet, you can run it in the emulator. Figure 14.1 shows the Contacts MIDlet as it appears when first run in the emulator.

FIGURE 14.1

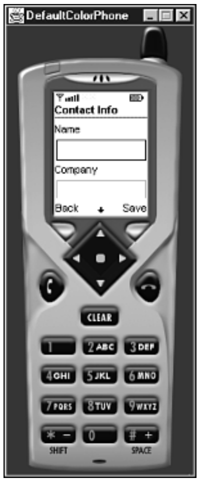
The Contacts MIDlet starts for the first time with an empty contact list.



Not surprisingly, the first time you run the Contacts MIDlet no contacts are in the main contact list. You can easily remedy this problem by adding some contacts. To add a contact, select the `Add` command. The Contact Info screen is then displayed, as shown in Figure 14.2.

FIGURE 14.2

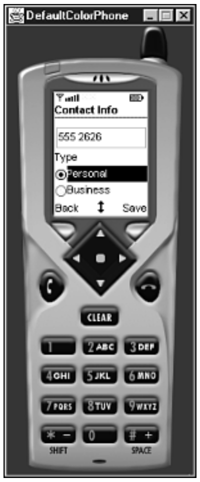
The Contact Info screen includes several input fields for entering the contact information.



Because the contact input fields don't fit on a single screen, you can use the arrow keys to scroll up and down and enter the contact information. Figure 14.3 shows contact information entered for a new contact.

FIGURE 14.3

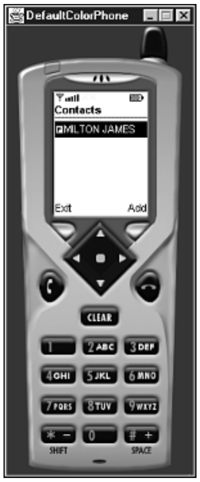
It is necessary to scroll around in the Contact Info screen to enter all the contact information.



After entering the information for the contact, you can select the `Save` command to add the contact to the database, or use the `Back` command to cancel adding the contact. Figure 14.4 shows the main contact list after selecting the `Save` command to add the new contact to the record store.

FIGURE 14.4

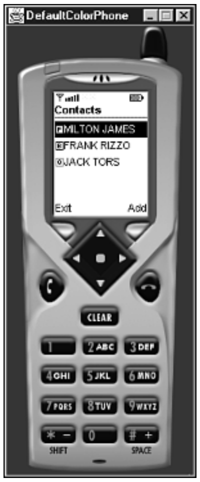
After saving a new contact, it appears in the main contact list with a small icon next to it to denote the contact type.



The contact name appears in the list along with a small graphical icon to the left of the name that indicates the contact type (Friend, Family, Business, and so forth). As you add more contacts, they appear in the contact list one after the next. Figure 14.5 shows the contact list after entering a few more contacts.

FIGURE 14.5

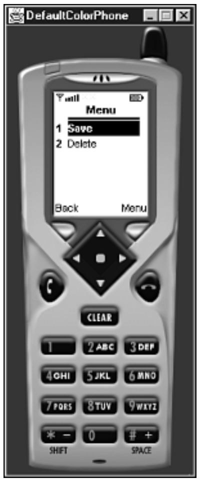
As you add more contacts, they are each added to the contact list on the main screen.



You edit and delete contacts using the Select button. To select a contact for editing or deletion, use the arrow keys to navigate to the desired contact and then press the Select button on the device. The contact will then be displayed in the Contact Info screen, where you can either edit it and select Save to save the changes, or select Delete to delete the contact. The `Save` and `Delete` commands are available through the Menu command, which is automatically added to the screen because only one button is available for the commands. Figure 14.6 shows how these commands are accessed through the `Menu` command.

FIGURE 14.6

The standard `Menu` *command is automatically added to the Contact Info screen to provide access to the* `Save` *and* `Delete` *commands.*



If you edit and save a contact, the changes to the contact are reflected in the main contact list. Likewise, if you delete a contact it is removed from the list. Figure 14.7 shows the main contact list after deleting the “FRANK RIZZO” contact.

That wraps up the test drive of the Contacts MIDlet, which demonstrates the main features of the MIDlet. I encourage you to spend some time tinkering with the MIDlet and assessing its usefulness as a mobile contact manager.

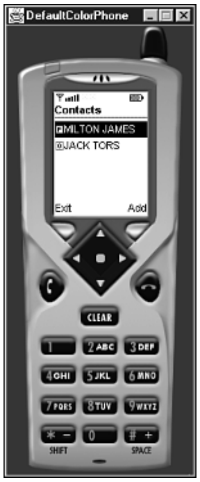


FIGURE 14.7
When you delete a contact it is removed from the main contact list.

Summary

This lesson applied the Record Management System (RMS) knowledge that you learned in the previous lesson to the development of a much more interesting MIDlet, a contact manager MIDlet. Although you didn't really learn any groundbreaking new MIDP or J2ME development concepts, you did learn how to create a practical MIDlet that could prove useful to MIDP device owners. Keep in mind that the ultimate goal of this book is to give you the skills to construct your own MIDlets from scratch.

This lesson began by exploring the potential need for a mobile contact manager that runs on MIDP devices. After you acquired an appreciation of why a contact manager might be a good thing to have on a mobile device, you jumped straight into the development of a contact manager MIDlet. After working through the code for the Contacts MIDlet, you tested it out in the J2ME emulator.

The next lesson continues to stay within the realm of practical MIDlets by leading you through the design and development of a personal finance MIDlet.

Q&A

Q Why aren't e-mail addresses included in the contact information in the Contacts MIDlet?

A The reason for this apparent oversight has to do more with keeping the MIDlet relatively simple than with it not making sense as a mobile contact manager feature. One small issue relating to adding e-mail information to a contact is that it's yet another piece of information that requires a lot of unwieldy typing. However, for those mobile devices that include e-mail services, it might come in handy to have the e-mail address stored for each contact. Keep in mind that it can also make sense to expand the Contacts MIDlet to include the mailing address for each contact.

Q Why is the `editing` member variable necessary?

A The `editing` member variable is necessary because the `Save` command doesn't inherently know whether the user is saving an edited contact or adding a new contact. An edited contact must set an existing record in the record store, whereas a new contact must be added to the record store. So, it is very important for the `Save` command code to know whether a contact is being edited or added to the record store. This knowledge is provided to the `Save` command code by the `editing` member variable, which serves as a flag that is set when a contact is edited.

Q When editing a contact, why are the `Save` and `Delete` commands displayed on a different screen?

A If you look closely at the device in the J2ME emulator, you'll notice that it only has two "soft" buttons that are available for performing MIDlet commands directly. Given the priority levels that were assigned to commands in the Contacts MIDlet, the `Back` command has priority over the `Save` and `Delete` commands, which means that it resides on one of the buttons. The `Save` and `Delete` commands have the same priority level, so there is no way to include them both when only one button is available. The MIDP solution is to place a command called `Menu` on the button that displays a screen containing the `Save` and `Delete` commands. The `Menu` command enables you to select commands using the number buttons instead of the "soft" buttons, so it serves as a good technique for providing access to multiple commands, even more than two.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. How is the contact information for a contact record stored?
2. How is the image icon for a contact associated with the contact in the main contact list?
3. How do you keep the user from selecting the `Delete` command while adding a contact?

Exercises

1. Modify the Contacts MIDlet so that e-mail and mailing address information are included in each contact. Hint: Make sure you delete the RMS database file before attempting to run the MIDlet with a different record format. If you don't, the MIDlet probably will read the records incorrectly.
2. Modify the Contacts MIDlet so that the contacts are organized in the main contact list according to their type. Hint: This involves sorting the contact list according to type as you enumerate the contacts in the record store. You must also factor in the insertion point of new contacts that are added so that they are placed with contacts of the appropriate type.

DAY 15

Managing Your Finances

Not long ago our society was primarily a cash society. Most day-to-day purchases were carried out with cash, along with an occasional check here and there. With the availability and heavy marketing of credit cards and debit cards, a considerable amount of daily transactions are now carried on without cash. If you are not extremely careful in keeping up with receipts, you can have problems when it comes to balancing your account at the end of the month. Therefore, it is useful to enter the transactions into a check register, which is what many people already do. However, a traditional paper check register doesn't help you determine what your balance is. That's where an electronic check register can really help.

This lesson guides you through the design and development of a check register MIDlet that enables you to keep track of financial transactions using your mobile phone, pager, or other MIDP device. In addition to being a good means of entering and storing financial transactions while you're on the go, the check register MIDlet also records a running balance so that you can see exactly how much money you have left in your account. The following major topics are covered throughout this lesson:

- Understanding the need for a mobile check register
- Constructing a mobile check register MIDlet
- Testing the check register MIDlet

Tracking Your Mobile Spending

Whether or not you are a fan of credit, there is no doubt that credit cards and debit cards add a certain degree of convenience to shopping. Even though studies have shown that people typically spend more freely when using plastic as opposed to cash, no one seems to be curtailing the use of plastic. Evidence of the continual rise of credit card and debit card use can clearly be seen in fast food restaurants, many of which now allow you to pay with plastic. When you think about it, credit cards and debit cards are a logical step forward in a world that is moving toward using less paper. The downside to not using paper to record transactions, however, is that you must be more vigilant with your personal record keeping.

Traditionally, people have used check registers to enter and track checking account transactions. The primary drawback to this kind of record keeping is that you must stop and calculate a new balance each time you enter a transaction, which is often a hassle. Some checkbook carrying cases have built-in calculators for this purpose. My solution is to forgo both the check register and the calculator and enter transactions directly into a MIDlet running on a mobile device. That's right: J2ME makes it possible to use your trusty mobile phone or pager as an electronic check register.

NEW TERM *electronic check register*—A software application that enables you to enter financial transactions associated with a bank account.

With a MIDlet check register, you not only have all of your transactions readily accessible in electronic form, but you also have a running balance that is continually calculated as you enter new transactions. Ideally, such a MIDlet would also somehow interface with a desktop financial application such as Quicken or Microsoft Money. That way you could synchronize your device to your desktop computer to automatically incorporate the mobile transactions with the account information in the financial software. The check register MIDlet you develop in this lesson isn't so ambitious as to provide support for connecting to a desktop financial application. Although this feature wouldn't necessarily be too difficult to implement, it nonetheless strays beyond the goals of this lesson.

The primary purpose of the check register MIDlet is to store individual transactions such as withdrawals and deposits. A significant part of the design of the MIDlet is determining exactly what information should be included in a transaction. The following list gives the main pieces of information associated with a financial transaction for the check register MIDlet:

- Transaction type (withdrawal or deposit)
- Number (check number)
- Date
- To/From
- Amount
- Memo

If you look at a paper check register, you will see similar spaces for entering this information. The transaction type is needed to determine whether the transaction is adding or removing funds from the account. The number is optional, and primarily applies to checks. The date for the transaction is fairly self-explanatory. The to/from information is the name of the person or business to which a check or debit is applied, or, in the case of a deposit, where the money is coming from. Finally, the memo is an optional piece of information to enter a quick note describing the transaction.

The job of the check register MIDlet is to provide a means of entering, editing, storing, and deleting transactions, along with calculating a running balance based on the transactions entered. Now that you understand what is required of the MIDlet and the data that it is to process, you can begin putting it together.

Building the CheckRegister MIDlet

Transactions in the CheckRegister MIDlet must be stored persistently, so you probably won't be too surprised to learn that the MIDlet manages transactions in much the same way that the ToDoList and Contacts MIDlets from the previous two lessons manage their data. This means that the CheckRegister MIDlet will rely on the RMS to persistently store and retrieve transactions. Even though the RMS is used in a familiar manner in the CheckRegister MIDlet, you will find that this MIDlet does some interesting new things when it comes to managing transactions. For example, the transaction list in the CheckRegister MIDlet always uses the last item in the list to store the balance of the check register. This requires some careful coding because the main list in the MIDlet serves two purposes: listing transactions and displaying the balance.

Managing the Transaction Records

Similar to the other RMS MIDlets that you have seen, the CheckRegister MIDlet uses two support classes to manage the persistent transaction data:

- **`Transaction`**—Represents a single transaction in the check register
- **`TransactionDB`**—Encapsulates the record store that stores transaction records

The following two sections explain these classes in detail. The `Transaction` class contains some interesting code related to the storage and formatting of different data types, and the `TransactionDB` class will look very familiar based upon the earlier RMS MIDlets.

The `Transaction` Class

There are several reasons why it is helpful to encapsulate a transaction in a class of its own. First, it is important to have the data that constitutes a transaction readily available in one place. More important, however, the `Transaction` class provides a convenient means of packing and unpacking transaction data for persistent storage and retrieval. Specifically, the `Transaction` class includes methods that pack the transaction data into a string for storage and unpack the data from a string for retrieval. Additionally, the `Transaction` class includes a few methods for formatting transaction data that isn't stored in a form suitable for display.

The complete code for the `Transaction` class is in Listing 15.1.

LISTING 15.1 The `Transaction` Class Represents a Single Financial Transaction

```
import java.util.*;

public class Transaction {
  private boolean withdrawal;
  private int num;
  private Date date;
  private String to;
  private int amount;  // in cents
  private String memo;

  public Transaction(boolean w, int n, Date dt, String t,
    String amt, String m) {
    withdrawal = w;
    num = n;
    date = dt;
    to = t;
    amount = parseAmount(amt);
    memo = m;
  }

  public Transaction(byte[] data) {
    unpack(new String(data));
  }

  public void unpack(String data) {
    String w;

    int start = 0, end = data.indexOf(';');
    w = data.substring(start, end);
```

LISTING 15.1 continued

```
    withdrawal = (w.compareTo("true") == 0);
    start = end + 1;
    end = data.indexOf(';', start);
    num = Integer.parseInt(data.substring(start, end));
    start = end + 1;
    end = data.indexOf(';', start);
    date = new Date(Long.parseLong(data.substring(start, end)));
    start = end + 1;
    end = data.indexOf(';', start);
    to = data.substring(start, end);
    start = end + 1;
    end = data.indexOf(';', start);
    amount = Integer.parseInt(data.substring(start, end));
    start = end + 1;
    memo = data.substring(start, data.length());
  }

  public String pack() {
    return (withdrawal ? "true":"false") + ';' +
      ((String)Integer.toString(num)) + ';' +
      ((String)Long.toString(date.getTime())) + ';' +
      to + ';' + ((String)Integer.toString(amount)) + ';' +
      memo;
  }

  public boolean isWithdrawal() {
    return withdrawal;
  }

  public int getNum() {
    return num;
  }

  public Date getDate() {
    return date;
  }

  public String getFormattedDate() {
    Calendar c = Calendar.getInstance();
    c.setTime(date);
    return Integer.toString(c.get(Calendar.MONTH)) + '/' +
      Integer.toString(c.get(Calendar.DAY_OF_MONTH));
  }

  public String getTo() {
    return to;
  }

  public int getAmount() {
```

LISTING 15.1 continued

```
    return amount;
  }

  public int getDollars() {
    return amount / 100;
  }

  public int getCents() {
    return amount % 100;
  }

  public String getFormattedAmount() {
    String c = Integer.toString(getCents());
    if (c.length() == 1)
      c = "0" + c;
    return new String(Integer.toString(getDollars()) + '.' + c);
  }

  private int parseAmount(String amt) {
    int dollars, cents;
    String s = new String(amt);

    // First parse the dollars
    if (amt.indexOf('.') != -1)
      s = amt.substring(0, amt.indexOf('.'));

    // Make sure there is a number left
    if (s.length() == 0)
      dollars = 0;
    else
      dollars = Integer.parseInt(s) * 100;

    // Parse the cents
    s = new String(amt);
    if (amt.indexOf('.') != -1)
      s = amt.substring(amt.indexOf('.') + 1, amt.length());
    else
      return dollars;

    // Make sure there is a number left, and no more than two
    if (s.length() == 0)
      return dollars;
    else if (s.length() > 2)
      s = s.substring(0, 2);
    cents = Integer.parseInt(s);

    return dollars + cents;
  }
```

LISTING 15.1 continued

```
    public String getMemo() {
      return memo;
    }

    public String getFormattedTransaction() {
      return getFormattedDate() + ' ' + to + " $" + getFormattedAmount();
    }
  }
```

Admittedly, this is a large class considering the fact that its main job is to encapsulate the data for a transaction. However, some very important tasks are to be carried out by methods in this class, as you will learn in just a moment. Before looking at the `Transaction` methods, take a quick look at the member variables for the class: `withdrawal`, `num`, `date`, `to`, `amount`, and `memo`. If you recall from the earlier discussion, these member variables directly correspond to the pieces of information required of a transaction. It is important to take note of the types of the member variables because it impacts how they are handled in the `Transaction` methods.

Another thing worth noting is the `amount` member variable, which is stored in cents. Remember that J2ME doesn't support floating-point math, so there is no way to store dollars and cents with a decimal point. The easy solution is to store all amounts as cents. The amount $15.50 would be stored in the `amount` variable as the integer 1550. Similarly, the amount $0.75 would be stored as the integer 75. This system works fine as long as you remember to take it into account whenever you're converting back and forth between the `amount` variable and a string that the user sees.

All of the `Transaction` member variables are initialized in the `Transaction()` constructors, the first of which accepts several values that each map to one of the member variables. The second constructor is used to create a `Transaction` object from a byte array of data, which is what is obtained when you read a transaction from a transaction record store using the RMS. This constructor calls the `unpack()` method, which is responsible for parsing through the packed transaction data and carefully extracting each piece of information. Similarly, the `pack()` method packs together all of the information associated with a transaction into a semicolon-delimited string. These two methods form the interface used to store and retrieve transactions to and from a record store, as you learn a little later in the lesson.

Most of the remaining methods in the `Transaction` class are access methods that are used to retrieve the different pieces of transaction information. A few of these methods are worth pointing out because they aren't exactly trivial. First, the `parseAmount()`

method is used to parse an integer amount (in cents) from a string that has the form “dollars.cents”. This method is called by the first `Transaction()` constructor to set the amount for the transaction. It is necessary because the user will be entering an amount as a string such as 54.32, which is 54 dollars and 32 cents. Of course, you already know that this needs to be converted to the integer 5432 according to our cent-based storage scheme. Although multiplying by 100 is sufficient for converting this example, what should happen when the user enters an amount such as 40? You must assume that they mean 40 dollars even though there is no decimal point. Another trickier example is if the user enters 12.5. In this case the user means $12.50—it is the job of the `parseAmount()` method to take all of these possibilities into account.

The `getFormattedDate()`, `getFormattedAmount()`, and `getFormattedTransaction()` methods are all used to format transaction information into a string form that can be displayed to the user. These methods will all prove to be extremely valuable when you get into the details of the CheckRegister MIDlet and begin displaying transaction information.

That sums up the most important aspects of the `Transaction` class, so let’s move on and learn about the `TransactionDB` class and how it supports a record store for transactions.

The `TransactionDB` Class

In the previous two lessons you developed record store classes for managing record stores containing individual records. Because good software development practices dictate reusing code whenever possible, I’m going to clue you in to the fact that the `TransactionDB` class is practically identical to the `ContactsDB` class from the previous lesson. Whereas the CheckRegister MIDlet is in many ways a very different MIDlet from the Contacts MIDlet, the general record store support that it requires is quite similar to that of the Contacts MIDlet. So, the following code for the `TransactionDB` class (see Listing 15.2) should look familiar even if the names have changed around a bit.

LISTING 15.2 The `TransactionDB` Class Is Responsible for Managing a Record Store of Transactions

```
import javax.microedition.rms.*;
import java.util.Enumeration;
import java.util.Vector;
import java.io.*;

public class TransactionDB {
  RecordStore recordStore = null;

  public TransactionDB(String name) {
    // Open the record store using the specified name
```

LISTING 15.2 continued

```
    try {
      recordStore = open(name);
    }
    catch(RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public RecordStore open(String fileName) throws RecordStoreException {
    return RecordStore.openRecordStore(fileName, true);
  }

  public void close() throws RecordStoreNotOpenException,
    RecordStoreException {
    // If the record store is empty, delete the file
    if (recordStore.getNumRecords() == 0) {
      String fileName = recordStore.getName();
      recordStore.closeRecordStore();
      recordStore.deleteRecordStore(fileName);
    }
    else {
      // Otherwise, close the record store
      recordStore.closeRecordStore();
    }
  }

  public Transaction getTransactionRecord(int id) {
    // Get the transaction record from the record store
    try {
      return (new Transaction(recordStore.getRecord(id)));
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }

    return null;
  }

  public synchronized void setTransactionRecord(int id, String record) {
    // Convert the string record to an array of bytes
    byte[] bytes = record.getBytes();

    // Set the record in the record store
    try {
      recordStore.setRecord(id, bytes, 0, bytes.length);
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }
  }
```

LISTING 15.2 continued

```
    public synchronized int addTransactionRecord(String record) {
      // Convert the string record to an array of bytes
      byte[] bytes = record.getBytes();

      // Add the byte array to the record store
      try {
        return recordStore.addRecord(bytes, 0, bytes.length);
      }
      catch (RecordStoreException e) {
        e.printStackTrace();
      }

      return -1;
    }

    public synchronized void deleteTransactionRecord(int id) {
      // Delete the transaction record from the record store
      try {
        recordStore.deleteRecord(id);
      }
      catch (RecordStoreException e) {
        e.printStackTrace();
      }
    }

    public synchronized RecordEnumeration enumerateTransactionRecords()
      throws RecordStoreNotOpenException {
      return recordStore.enumerateRecords(null, null, false);
    }
  }
```

You already have a good feel for how this class works, so I'll spare you an in-depth discussion of each of the methods. Instead, the following list summarizes the purpose of each method in the `TransactionDB` class:

- **`open()`**—Opens the transaction record store
- **`close()`**—Closes the transaction record store
- **`getTransactionRecord()`**—Retrieves a transaction record from the record store
- **`setTransactionRecord()`**—Alters an existing transaction record in the record store by setting it with new transaction data
- **`addTransactionRecord()`**—Adds a new transaction record to the record store
- **`deleteTransactionRecord()`**—Deletes a transaction record from the record store
- **`enumerateTransactionRecords()`**—Obtains an enumeration of the transaction records in the record store

Hopefully this quick explanation of the TransactionDB methods is enough to get you up-to-speed with how the class is used to help manage the transaction record store. You should now be ready to tackle the code for the actual MIDlet, which is where things get much more interesting.

Assessing the Member Variables

As you have no doubt learned throughout the book thus far, the best place to start constructing a MIDlet is to define its member variables. In the case of the CheckRegister MIDlet, the member variables not only store the user interface for the MIDlet, but they also control the MIDlet's behavior when it comes to successfully managing the transactions. The following are the member variables for the CheckRegister MIDlet:

```
private Command exitCommand, addCommand, backCommand, saveCommand,
  deleteCommand;
private Display display;
private List mainScreen;
private Form transactionScreen;
private ChoiceGroup transactionType;
private TextField numField, toField, amountField, memoField;
private DateField dateField;
private boolean editing = false;
private TransactionDB db = null;
private Vector transactionIDs = new Vector();
private Image[] typeImages;
private int balanceAmt = 0;
private int balanceIndex;
private int transactionAmt = 0;
private int transactionNum = 0;
```

Many of these member variables should be somewhat familiar to you because they represent parts of the MIDlet's user interface. For example, the first few variables represent the commands for the MIDlet, which enable you to interact with transactions and exit the MIDlet. The Exit and Add commands appear on the main screen, while the Back, Save, and Delete commands appear on the transaction form screen. Selecting the Add command displays the transaction form screen, which enables the user to enter a new transaction. The new transaction can then be saved to the record store by selecting the Save command, or cancelled by using the Back command. To edit a transaction, use the Select button on the device, which issues the standard Select command. When editing a transaction, as opposed to adding a new one, the Delete command is added to the transaction screen so that the user has the ability to delete the transaction.

The main screen of the MIDlet is a list that contains all of the transactions along with the balance of the check register. The transaction screen is a form that includes various GUI components for entering the specifics of a transaction. A choice group enables the user to choose whether the transaction is a withdrawal or a deposit. Text fields enable the user to enter the number, to/from, amount, and memo information for the transaction. Finally, the date for the transaction can be entered into the date field, which provides its own user interface for selecting dates.

The `editing` member variable is used by the MIDlet to determine whether the user is adding or editing a transaction. This is very important because the transaction screen is used for both purposes, and you need to know whether to add a new transaction to the record store or replace an existing one.

The `db` member variable stores the record store manager class, which you learned about in the previous section. The `transactionIDs` member variable is a vector that keeps track of the record ID for each transaction in the check register. If you recall from the previous RMS MIDlets, this variable is necessary because the record ID is the only association between a transaction and the record store.

The `typeImages` member variable stores an array of images that are used to provide a graphical icon for the transaction type (withdrawal or deposit). The withdrawal icon is a small red down arrow, and the deposit icon is a small black up arrow. One of these icons will be displayed next to each transaction in the check register to provide a visual cue regarding withdrawals and deposits.

The `balanceAmt` member variable stores the total balance of the check register, and is displayed at the bottom of the transaction list. This variable must be updated any time a transaction is added, edited, or deleted from the check register. The `balanceIndex` member variable stores the index of the balance in the transaction list. This variable is important because as you add and delete records the balance will shift up and down in the list. Because the balance is constantly changing, it must also constantly be updated, which requires that you know the current index of it in the list.

The `transactionAmt` member variable is related to the `balanceAmt` variable, and reveals a tricky aspect of the CheckRegister MIDlet. The way that you edit a transaction is by replacing it in the transaction list. At the point in the MIDlet's code where you save an edited transaction you don't have access to the amount of the transaction prior to it being edited. You might wonder why you would need this information. The balance of the check register is already calculated to include the old transaction amount. So, to correctly apply the new amount you must first adjust the balance based on the old transaction amount. Because you don't have access to this information at the point where you need it, the `transactionAmt` member variable is used to store it for you.

The last member variable in the CheckRegister MIDlet is `transactionNum`, which adds a nifty little user-friendly feature to the MIDlet. If you recall, the number of a transaction typically identifies the number for a check. Because checks are numbered incrementally, it helps the user if you automatically keep up with the highest numbered transaction, and then automatically set the default number for subsequent transactions to one greater than that. So, if the user enters check number 145 and then creates another transaction, the new transaction will automatically be numbered 146 unless the user overrides it. Thoughtful features like this help make a MIDlet a worthwhile convenience.

Inside the Constructor

The constructor for the CheckRegister does quite a lot of work, and is therefore an important part of the MIDlet's code. In addition to initializing member variables, the `CheckRegister()` constructor handles all of the details involved in setting up the MIDlet's graphical user interface. Of course, this also involves reading all the transactions from the record store and displaying them in the transaction list, not to mention calculating and displaying a balance. Listing 15.3 supplies the code for the `CheckRegister()` constructor.

LISTING 15.3 The `CheckRegister()` Constructor Takes Care of Initializing the MIDlet

```
public CheckRegister() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  addCommand = new Command("Add", Command.SCREEN, 3);
  backCommand = new Command("Back", Command.BACK, 2);
  saveCommand = new Command("Save", Command.OK, 3);
  deleteCommand = new Command("Delete", Command.SCREEN, 3);

  // Create the main screen
  mainScreen = new List("Transactions", List.IMPLICIT);

  // Set the Exit and Add commands for the main screen
  mainScreen.addCommand(exitCommand);
  mainScreen.addCommand(addCommand);
  mainScreen.setCommandListener(this);

  // Create the transaction screen
  transactionScreen = new Form("Transaction Info");
  String[] choices = { "Withdrawal", "Deposit" };
  transactionType = new ChoiceGroup("", ChoiceGroup.EXCLUSIVE, choices, null);
  transactionScreen.append(transactionType);
  numField = new TextField("Number", "", 6, TextField.NUMERIC);
  transactionScreen.append(numField);
  dateField = new DateField("Date", DateField.DATE);
```

LISTING 15.3 continued

```
transactionScreen.append(dateField);
toField = new TextField("To/From", "", 16, TextField.ANY);
transactionScreen.append(toField);
amountField = new TextField("Amount", "", 12, TextField.ANY);
transactionScreen.append(amountField);
memoField = new TextField("Memo", "", 20, TextField.ANY);
transactionScreen.append(memoField);

// Set the Back, Save, and Delete commands for the transaction screen
transactionScreen.addCommand(backCommand);
transactionScreen.addCommand(saveCommand);
transactionScreen.addCommand(deleteCommand);
transactionScreen.setCommandListener(this);

// Load the type images
try {
  typeImages = new Image[2];
  typeImages[0] = Image.createImage("/Withdrawal.png");
  typeImages[1] = Image.createImage("/Deposit.png");
}
catch (IOException e) {
  System.err.println("EXCEPTION: Failed loading images!");
}

// Open the transaction database
try {
  db = new TransactionDB("transactions");
}
catch(Exception e) {
  System.err.println("EXCEPTION: Problem opening the database.");
}

// Read through the database and build a list of record IDs
RecordEnumeration records = null;
try {
  records = db.enumerateTransactionRecords();
  while(records.hasNextElement())
    transactionIDs.addElement(new Integer(records.nextRecordId()));
}
catch(Exception e) {
  System.err.println("EXCEPTION: Problem reading the transaction records.");
}

// Read through the database and fill the transaction list
// Also update the balance and increment the transaction number
records.reset();
try {
  while(records.hasNextElement()) {
```

LISTING 15.3 continued

```
      Transaction transaction = new Transaction(records.nextRecord());
      mainScreen.append(transaction.getFormattedTransaction(),
        typeImages[transaction.isWithdrawal() ? 0:1]);
      if (transaction.isWithdrawal())
        balanceAmt -= transaction.getAmount();
      else
        balanceAmt += transaction.getAmount();
      if (transaction.getNum() > transactionNum)
        transactionNum = transaction.getNum();
    }
  }
  catch(Exception e) {
    System.err.println("EXCEPTION: Problem reading the transaction records.");
  }

  // Format and add the balance to the list
  balanceIndex = mainScreen.append(formatBalance(), null);
}
```

The `CheckRegister()` constructor handles all of the initialization details for the MIDlet, including setting up the user interface and reading the transaction records from the record store. The constructor begins by obtaining the `Display` object for the MIDlet, and then creating the commands. The main list screen is created next, followed by the transaction screen, and commands are added to each. The transaction screen contains several fields for entering the different pieces of transaction information.

After setting up the screens, the `CheckRegister()` constructor moves on to open the record store database and build a vector of transaction record IDs. The constructor then resets the record store enumeration and steps through the records one more time. The second time through the records involves adding the transactions to the transaction list and calculating a running balance for the entire check register. Additionally, the transaction number is updated to reflect the highest numbered transaction. After stepping through all of the transactions and adding them to the transaction list, the balance is formatted and added as the last item in the list.

That wraps up the `CheckRegister()` constructor. The remaining code of significance in the MIDlet is found in the `commandAction()` method, which you learn about next.

Handling Commands

Because commands drive the operation of the CheckRegister MIDlet, it shouldn't be too terribly surprising that the `commandAction()` method includes most of the MIDlet's functionality. This method is responsible for handling all of the MIDlet's commands, including the `Exit`, `Add`, `Back`, `Save`, and `Delete` commands. I think you have a good idea how the `Exit` command works, so I'll move on and say that the `Add` command is used to add a new transaction, and the `Save` command is used to save a transaction that is either being edited or added. The `Back` command cancels out of the editing or adding of a transaction. Finally, the `Delete` command is only available while editing a transaction, and results in the transaction being deleted from the transaction list and the record store. One other command handled by the `commandAction()` method is the special `Select` command, which is carried out when the user presses the Select button on the device. This command enables you to edit the currently selected transaction.

Listing 15.4 represents the complete code for the `commandAction()` method, which reveals how each of the MIDlet commands are handled.

LISTING 15.4 The `commandAction()` Method Responds to Commands in the MIDlet

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == addCommand) {
    // Clear the transaction fields
    transactionType.setSelectedIndex(0, true);
    numField.setString(Integer.toString(transactionNum));
    dateField.setDate(Calendar.getInstance().getTime());
    toField.setString("");
    amountField.setString("");
    memoField.setString("");

    // Remove the Delete command from the transaction screen
    transactionScreen.removeCommand(deleteCommand);
    editing = false;

    // Set the current display to the transaction screen
    display.setCurrent(transactionScreen);
  }
  else if (c == List.SELECT_COMMAND) {
    // Allow an edit as long as this isn't the balance item
    if (mainScreen.getSelectedIndex() != (mainScreen.size() - 1)) {
      // Get the record ID of the currently selected transaction
      int index = mainScreen.getSelectedIndex();
      int id = ((Integer)transactionIDs.elementAt(index)).intValue();

      // Retrieve the transaction record from the database
      Transaction transaction = db.getTransactionRecord(id);

      // Initialize the transaction fields
      transactionType.setSelectedIndex(transaction.isWithdrawal() ? 0:1, true);
```

LISTING 15.4 continued

```
      numField.setString(Integer.toString(transaction.getNum()));
      dateField.setDate(transaction.getDate());
      toField.setString(transaction.getTo());
      amountField.setString(transaction.getFormattedAmount());
      memoField.setString(transaction.getMemo());

      // Save the transaction amount
      transactionAmt = transaction.isWithdrawal() ?
        transaction.getAmount():(-transaction.getAmount());

      // Add the Delete command to the transaction screen
      transactionScreen.addCommand(deleteCommand);
      editing = true;

      // Set the current display to the transaction screen
      display.setCurrent(transactionScreen);
    }
  }
  else if (c == deleteCommand) {
    // Get the record ID of the currently selected transaction
    int index = mainScreen.getSelectedIndex();
    int id = ((Integer)transactionIDs.elementAt(index)).intValue();

    // Retrieve the transaction record from the database
    Transaction transaction = db.getTransactionRecord(id);

    // Update the balance
    if (transaction.isWithdrawal())
      balanceAmt += transaction.getAmount();
    else
      balanceAmt -= transaction.getAmount();

    // Format and update the balance to the list
    mainScreen.set(balanceIndex, formatBalance(), null);

    // Delete the transaction record
    db.deleteTransactionRecord(id);
    transactionIDs.removeElementAt(index);
    mainScreen.delete(index);

    // Decrement the balance index
    balanceIndex--;

    // Set the current display back to the main screen
    display.setCurrent(mainScreen);
  }
  else if (c == backCommand) {
    // Set the current display back to the main screen
    display.setCurrent(mainScreen);
  }
```

LISTING 15.4 continued

```
    else if (c == saveCommand) {
      // Create a record for the transaction
      String n = numField.getString();
      Transaction transaction = new Transaction(
        (transactionType.getSelectedIndex() == 0) ? true:false,
        (n.length() == 0) ? 0:Integer.parseInt(n),
        dateField.getDate(), toField.getString(),
        amountField.getString(), memoField.getString());

      // Update the balance
      if (transaction.isWithdrawal())
        balanceAmt -= transaction.getAmount();
      else
        balanceAmt += transaction.getAmount();

      if (editing) {
        // Modify the balance to account for the old transaction amount
        balanceAmt += transactionAmt;

        // Get the record ID of the currently selected transaction
        int index = mainScreen.getSelectedIndex();
        int id = ((Integer)transactionIDs.elementAt(index)).intValue();

        // Set the transaction in the database
        db.setTransactionRecord(id, transaction.pack());
        mainScreen.set(index, transaction.getFormattedTransaction(),
          typeImages[transaction.isWithdrawal() ? 0:1]);
      }
      else {
        // Add the transaction to the database
        transactionIDs.addElement(new Integer(db.addTransactionRecord(
➥transaction.pack())));
        mainScreen.insert(balanceIndex, transaction.getFormattedTransaction(),
          typeImages[transaction.isWithdrawal() ? 0:1]);

        // Increment the balance index
        balanceIndex++;

        // Increment the transaction number
        transactionNum = transaction.getNum() + 1;
      }

      // Format and update the balance to the list
      mainScreen.set(balanceIndex, formatBalance(), null);

      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
  }
```

This is a fairly hefty chunk of code, but don't forget that the bulk of the CheckRegister MIDlet's functionality is carried out in this method alone. The method begins by handling the `Exit` command, which relies on standard exit code used in all MIDlets. The `Add` command is then handled, and involves clearing the fields on the transaction form and removing the `Delete` command. Notice that the transaction number is used to set the default transaction number for the transaction, and the current date is set as the default date. The transaction screen is then displayed so that the user can enter the transaction information. It's important to notice that the `editing` flag is set to `false` to indicate that the user is adding a new transaction, and not editing an existing one.

The `Select` command is next on the agenda, and takes care of editing an existing transaction. Before doing anything, the code for the Select code first checks to make sure that the currently selected item in the transaction list isn't the balance item. It obviously doesn't make any sense to edit the balance because it is automatically calculated, so this check is important. If a valid transaction is indeed selected in the list, the `Select` command code uses the currently selected transaction as the basis for initializing the fields in the transaction form. Of course, this requires the use of the ID of the transaction, which is stored in the `transactionIDs` member variable.

After obtaining the record ID, the Select handler code reads the record from the record store and creates a `Transaction` object. This object is then used to set the fields of the transaction form for editing. At this point the transaction amount is stored in the `transactionAmt` member variable so that the balance of the check register can be properly calculated later if the user changes the transaction and saves it. The `Delete` command is then added to the transaction screen so that the user has the option of deleting the transaction, and the `editing` flag is set to `true`.

The `Delete` command code is responsible for deleting the currently selected transaction from the transaction list and the record store. Similar to the `Select` command code, the Delete code uses the record ID to read the transaction from the record store. This is necessary because you must alter the balance of the check register because of the removal of the transaction, which requires knowing the transaction amount. After updating the balance amount, the new balance is updated in the transaction list. The transaction record is then deleted from both the transaction list and record store. The Delete handler code then finishes by decrementing the index of the balance, which is very important because the balance's index in the list changes because of the deleted transaction.

The `Back` command code is very simple because all it has to do is set the current display to the main transaction list screen. There isn't much more to say about this code, so let's move on to the `Save` command.

The code for the `Save` command accomplishes two tasks: It saves an edited transaction or a newly added transaction. The determination of which task to perform is made by examining the `editing` flag. Prior to checking this flag, however, the `Save` command code creates a `Transaction` object and initializes it using the information stored in the transaction form fields. The check register balance is then updated to reflect the transaction. Most of the remaining Save command code is split into two parts: the code for saving an edited transaction and the code for adding a new transaction.

Before saving an edited transaction, it is important to alter the balance so that the old transaction amount is no longer factored into the balance. After taking care of that bit of business, the `Save` command code uses the `Transaction` object created earlier in the handler code to set the record in the record set. The formatted transaction in the transaction list is also updated to reflect the newly edited transaction information. The code that adds a new transaction is similar to the "edit" code except that it adds the new transaction to the record store and transaction list. The "add" code also increments the balance index because of the addition of the new transaction in the list, and also adjusts the transaction number so that it is one higher than the new transaction's number.

After carrying out the saving of the transaction, the `Save` command code finishes by updating the balance in the transaction list, and then setting the main list screen as the current display. Although the `commandAction()` method packs in a lot of code, it is relatively straightforward if you take it a step at a time. To make things a little cleaner, you could easily break apart each command handler into its own method. However, for the purposes of this discussion it made sense just to leave it as a single method.

In a few places in the code for the `CheckRegister()` constructor and the `commandAction()` method, a method named `formatBalance()` is called to obtain a formatted string containing the balance of the check register. This method takes the current balance, specified in cents, and formats it into a string that is suitable for being displayed in the transaction list. As an example, if the balance is set to the integer 52548 then the `formatBalance()` method will return the string `"Balance : $525.48"`. This is the string that will be displayed at the bottom of the transaction list. The code for the `formatBalance()` method is as follows:

```
private String formatBalance() {
  String c = Integer.toString(Math.abs(balanceAmt % 100));
  if (c.length() == 1)
    c = "0" + c;
  return "Balance : $" + Integer.toString(balanceAmt / 100) + '.' + c;
}
```

As you can see, the dollars and cents of the balance are extracted from the `balanceAmt` member variable and formatted into the complete balance string.

Putting It All Together

You've now seen the most important pieces of the puzzle in regard to the CheckRegister MIDlet. So you can see how all of the code for the MIDlet fits together, check out the following complete source code for the `CheckRegister` MIDlet class (see Listing 15.5).

LISTING 15.5 The Complete Source Code for the `CheckRegister` MIDlet Class

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;

public class CheckRegister extends MIDlet implements CommandListener {
  private Command exitCommand, addCommand, backCommand, saveCommand,
    deleteCommand;
  private Display display;
  private List mainScreen;
  private Form transactionScreen;
  private ChoiceGroup transactionType;
  private TextField numField, toField, amountField, memoField;
  private DateField dateField;
  private boolean editing = false;
  private TransactionDB db = null;
  private Vector transactionIDs = new Vector();
  private Image[] typeImages;
  private int balanceAmt = 0;
  private int balanceIndex;
  private int transactionAmt = 0;
  private int transactionNum = 0;

  public CheckRegister() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    addCommand = new Command("Add", Command.SCREEN, 3);
    backCommand = new Command("Back", Command.BACK, 2);
    saveCommand = new Command("Save", Command.OK, 3);
    deleteCommand = new Command("Delete", Command.SCREEN, 3);

    // Create the main screen
    mainScreen = new List("Transactions", List.IMPLICIT);

    // Set the Exit and Add commands for the main screen
    mainScreen.addCommand(exitCommand);
```

LISTING 15.5 continued

```
    mainScreen.addCommand(addCommand);
    mainScreen.setCommandListener(this);

    // Create the transaction screen
    transactionScreen = new Form("Transaction Info");
    String[] choices = { "Withdrawal", "Deposit" };
    transactionType = new ChoiceGroup("", ChoiceGroup.EXCLUSIVE, choices, null);
    transactionScreen.append(transactionType);
    numField = new TextField("Number", "", 6, TextField.NUMERIC);
    transactionScreen.append(numField);
    dateField = new DateField("Date", DateField.DATE);
    transactionScreen.append(dateField);
    toField = new TextField("To/From", "", 16, TextField.ANY);
    transactionScreen.append(toField);
    amountField = new TextField("Amount", "", 12, TextField.ANY);
    transactionScreen.append(amountField);
    memoField = new TextField("Memo", "", 20, TextField.ANY);
    transactionScreen.append(memoField);

    // Set the Back, Save, and Delete commands for the transaction screen
    transactionScreen.addCommand(backCommand);
    transactionScreen.addCommand(saveCommand);
    transactionScreen.addCommand(deleteCommand);
    transactionScreen.setCommandListener(this);

    // Load the type images
    try {
      typeImages = new Image[2];
      typeImages[0] = Image.createImage("/Withdrawal.png");
      typeImages[1] = Image.createImage("/Deposit.png");
    }
    catch (IOException e) {
      System.err.println("EXCEPTION: Failed loading images!");
    }

    // Open the transaction database
    try {
      db = new TransactionDB("transactions");
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem opening the database.");
    }

    // Read through the database and build a list of record IDs
    RecordEnumeration records = null;
    try {
      records = db.enumerateTransactionRecords();
      while(records.hasNextElement())
        transactionIDs.addElement(new Integer(records.nextRecordId()));
```

LISTING 15.5 continued

```
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the transaction records.");
    }

    // Read through the database and fill the transaction list
    // Also update the balance and increment the transaction number
    records.reset();
    try {
      while(records.hasNextElement()) {
        Transaction transaction = new Transaction(records.nextRecord());
        mainScreen.append(transaction.getFormattedTransaction(),
          typeImages[transaction.isWithdrawal() ? 0:1]);
        if (transaction.isWithdrawal())
          balanceAmt -= transaction.getAmount();
        else
          balanceAmt += transaction.getAmount();
        if (transaction.getNum() > transactionNum)
          transactionNum = transaction.getNum();
      }
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the transaction records.");
    }

    // Format and add the balance to the list
    balanceIndex = mainScreen.append(formatBalance(), null);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the main screen
    display.setCurrent(mainScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
    // Close the transaction database
    try {
      db.close();
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem closing the database.");
    }
  }

  public void commandAction(Command c, Displayable s) {
```

LISTING 15.5 continued

```
        if (c == exitCommand) {
          destroyApp(false);
          notifyDestroyed();
        }
        else if (c == addCommand) {
          // Clear the transaction fields
          transactionType.setSelectedIndex(0, true);
          numField.setString(Integer.toString(transactionNum));
          dateField.setDate(Calendar.getInstance().getTime());
          toField.setString("");
          amountField.setString("");
          memoField.setString("");

          // Remove the Delete command from the transaction screen
          transactionScreen.removeCommand(deleteCommand);
          editing = false;

          // Set the current display to the transaction screen
          display.setCurrent(transactionScreen);
        }
        else if (c == List.SELECT_COMMAND) {
          // Allow an edit as long as this isn't the balance item
          if (mainScreen.getSelectedIndex() != (mainScreen.size() - 1)) {
            // Get the record ID of the currently selected transaction
            int index = mainScreen.getSelectedIndex();
            int id = ((Integer)transactionIDs.elementAt(index)).intValue();

            // Retrieve the transaction record from the database
            Transaction transaction = db.getTransactionRecord(id);

            // Initialize the transaction fields
            transactionType.setSelectedIndex(transaction.isWithdrawal() ? 0:1,
➥true);
            numField.setString(Integer.toString(transaction.getNum()));
            dateField.setDate(transaction.getDate());
            toField.setString(transaction.getTo());
            amountField.setString(transaction.getFormattedAmount());
            memoField.setString(transaction.getMemo());

            // Save the transaction amount
            transactionAmt = transaction.isWithdrawal() ?
              transaction.getAmount():(-transaction.getAmount());

            // Add the Delete command to the transaction screen
            transactionScreen.addCommand(deleteCommand);
            editing = true;

            // Set the current display to the transaction screen
```

LISTING 15.5 continued

```
        display.setCurrent(transactionScreen);
      }
    }
    else if (c == deleteCommand) {
      // Get the record ID of the currently selected transaction
      int index = mainScreen.getSelectedIndex();
      int id = ((Integer)transactionIDs.elementAt(index)).intValue();

      // Retrieve the transaction record from the database
      Transaction transaction = db.getTransactionRecord(id);

      // Update the balance
      if (transaction.isWithdrawal())
        balanceAmt += transaction.getAmount();
      else
        balanceAmt -= transaction.getAmount();

      // Format and update the balance to the list
      mainScreen.set(balanceIndex, formatBalance(), null);

      // Delete the transaction record
      db.deleteTransactionRecord(id);
      transactionIDs.removeElementAt(index);
      mainScreen.delete(index);

      // Decrement the balance index
      balanceIndex--;

      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
    else if (c == backCommand) {
      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
    else if (c == saveCommand) {
      // Create a record for the transaction
      String n = numField.getString();
      Transaction transaction = new Transaction(
        (transactionType.getSelectedIndex() == 0) ? true:false,
        (n.length() == 0) ? 0:Integer.parseInt(n),
        dateField.getDate(), toField.getString(),
        amountField.getString(), memoField.getString());

      // Update the balance
      if (transaction.isWithdrawal())
        balanceAmt -= transaction.getAmount();
      else
```

LISTING 15.5 continued

```
          balanceAmt += transaction.getAmount();

        if (editing) {
          // Modify the balance to account for the old transaction amount
          balanceAmt += transactionAmt;

          // Get the record ID of the currently selected transaction
          int index = mainScreen.getSelectedIndex();
          int id = ((Integer)transactionIDs.elementAt(index)).intValue();

          // Set the transaction in the database
          db.setTransactionRecord(id, transaction.pack());
          mainScreen.set(index, transaction.getFormattedTransaction(),
            typeImages[transaction.isWithdrawal() ? 0:1]);
        }
        else {
          // Add the transaction to the database
          transactionIDs.addElement(new Integer(db.addTransactionRecord(
➥transaction.pack())));
          mainScreen.insert(balanceIndex, transaction.getFormattedTransaction(),
            typeImages[transaction.isWithdrawal() ? 0:1]);

          // Increment the balance index
          balanceIndex++;

          // Increment the transaction number
          transactionNum = transaction.getNum() + 1;
        }

        // Format and update the balance to the list
        mainScreen.set(balanceIndex, formatBalance(), null);

        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
    }

    private String formatBalance() {
      String c = Integer.toString(Math.abs(balanceAmt % 100));
      if (c.length() == 1)
        c = "0" + c;
      return "Balance : $" + Integer.toString(balanceAmt / 100) + '.' + c;
    }
  }
```

This code listing reiterates the fact that most of the functionality of the CheckRegister MIDlet is contained within the `CheckRegister()` constructor and the `commandAction()` method. This code represents your most ambitious MIDlet thus far in the book when you consider that it manages a fairly intricate user interface and also stores and retrieves persistent data. The MIDlet certainly isn't perfect, but it has all of the basic functionality you might expect in a mobile electronic check register. Let's now take a look at how it works by taking it for a spin in the J2ME emulator.

Testing the CheckRegister MIDlet

As you know, the CheckRegister MIDlet is designed to provide a means of entering financial transactions into an electronic check register than runs on a mobile device such as a phone or pager. The MIDlet starts out with a balance of zero, which means you should probably begin with a deposit so that money is available. Figure 15.1 shows the CheckRegister MIDlet upon startup in the J2ME emulator, with a balance of $0.00.

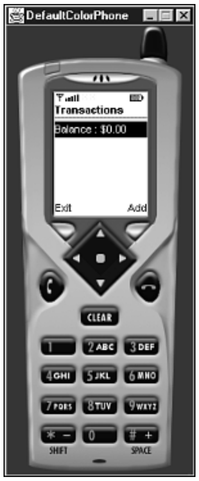


FIGURE 15.1
The CheckRegister MIDlet begins with a balance of $0.00 in the main transaction list.

To get started with the MIDlet, it is best to enter a deposit so that you have some funds with which to work. Select the `Add` command to add a new transaction to the transaction list. Figure 15.2 shows the Transaction Info screen, which is where you enter new transactions.

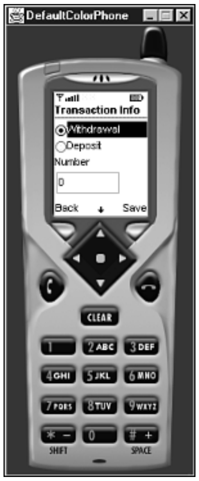


FIGURE 15.2

The Transaction Info screen includes several input fields for entering transaction information.

As the figure shows, so much information is required for a transaction that it doesn't all fit on a single screen in the emulator. You can use the arrow keys to move up and down the transaction screen and enter all of the relevant pieces of transaction information. Figure 15.3 shows part of a deposit transaction that has been entered.

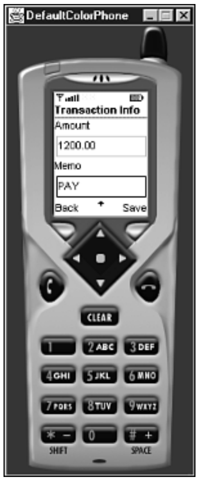


FIGURE 15.3

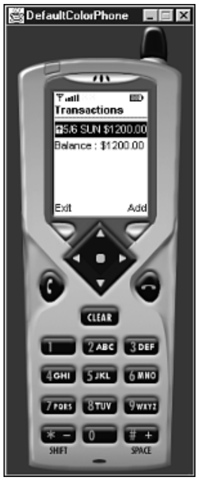
Entering a transaction in the Transaction Info screen involves scrolling up and down in the screen to access all of the input fields.

After you've finished entering the transaction information, select the `Save` command. The new transaction is added to the main transaction list and the record store.

Figure 15.4 shows the main transaction list with the new transaction added, as well as the new check register balance.

FIGURE 15.4

After saving a new transaction, it appears in the main transaction list along with the updated check register balance.

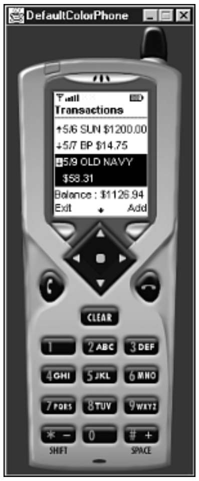


```
Transactions
5/6 SUN $1200.00
Balance : $1200.00
Exit                 Add
```

If you look closely at the figure you'll see a small graphical icon to the left of the transaction. This icon is a small black up arrow that indicates that the transaction is a deposit; withdrawals use a small red down arrow icon. If you continue to add transactions, they will be added to the transaction list. The balance is also constantly updated to reflect the new transactions. Figure 15.5 shows the transaction list after entering a few more transactions.

FIGURE 15.5

As you add more transactions, they are each added to the transaction list, and the balance is continually updated.



```
Transactions
↑5/6 SUN $1200.00
↓5/7 BP $14.75
 5/9 OLD NAVY
  $58.31
Balance : $1126.94
Exit        ↓        Add
```

At some point you might realize that an error was made or that you need to edit a transaction for some other reason. The Select button enables you to edit the currently selected transaction; just navigate to the transaction in the list and press the Select button on the device. The Transaction Info screen is then displayed with the transaction information loaded into the input fields. From this screen you can edit the transaction and select Save to save the changes or `Delete` to delete the transaction. The `Save` and `Delete` commands are available by selecting the `Menu` command, which is automatically added to the screen because only one button is available for the commands. Figure 15.6 shows how these commands are accessed with the `Menu` command.

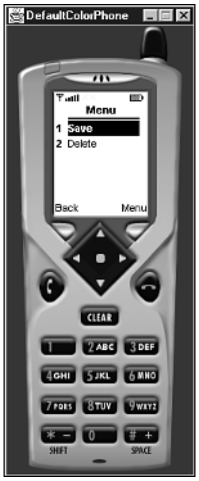
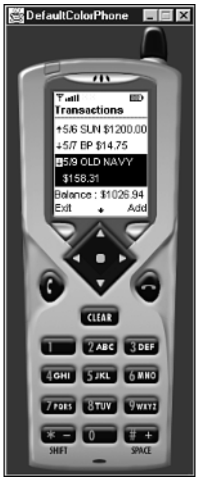


FIGURE 15.6
The standard Menu command is automatically added to the Transaction Info screen to provide access to the Save and Delete commands.

Remember that when you edit the type or amount of a transaction it impacts the balance of the check register. So, in addition to the transaction changing in the transaction list, the entire balance of the check register is also changed to reflect the transaction changes. Figure 15.7 shows the transaction list after editing the “OLD NAVY” transaction so that its amount is $158.31 instead of $58.31.

FIGURE 15.7

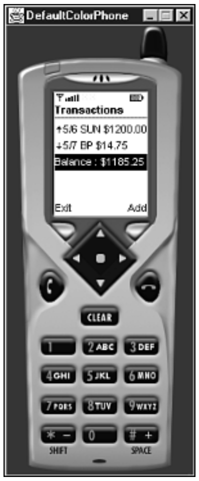
When you change the amount of a transaction, the transaction is changed in the transaction list, along with the balance.



Of course, deleting a transaction also alters the transaction list and the balance of the check register. Figure 15.8 shows how the check register changes when "OLD NAVY" transaction is deleted.

FIGURE 15.8

Deleting a transaction removes it from the transaction list and also changes the balance.



As you can see, the check register MIDlet could be quite handy at keeping track of financial transactions when you are on the go. It would be nice if it could somehow interface with a desktop financial application such as Quicken or Microsoft Money, but that's a little beyond that scope of this lesson.

Summary

This lesson guided you through the design and development of a practical financial MIDlet that enables you to enter transactions into a mobile device much as you would enter them into a paper check register. As the world continues to move toward using less paper, it is only logical that you should consider keeping up with your financial transactions electronically. Many of us already hold on to receipts so that we can enter them into a financial application when we get home. However, the CheckRegister MIDlet makes it possible to enter transactions directly into your mobile device while you're out shopping, which is a significant benefit over trying to hang on to receipts.

The next lesson wraps up this part of the book with another practical MIDlet that is used to manage persistent information. The MIDlet in the next lesson is an auction MIDlet that enables you to keep track of eBay auctions from the convenience of your mobile device.

Q&A

Q Why doesn't the CheckRegister MIDlet allow you to categorize transactions?

A There are two reasons why the CheckRegister MIDlet doesn't support categorized transactions. First, categories would add unnecessary complication to the MIDlet that in many ways would get in the way of explaining the basics of how the MIDlet functions. Second, it takes a decent amount of effort to set up categories, which is something most people probably won't have the desire to do in a mobile setting. Ideally, it might be neat if the MIDlet could synchronize with a desktop financial application and acquire categories from it. However, this task is a little too ambitious for this lesson.

Q Could the CheckRegister MIDlet be modified to support multiple accounts?

A Absolutely. Although it would add some complexity to the MIDlet, it would actually be logical to support multiple accounts. You could then add a field to each transaction that indicates to which account it belongs. The user interface would probably need to pick up an additional screen that allows you to select from between multiple accounts. This screen would also allow you to add, edit, and delete entire accounts, including the transactions associated with them.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What are the main benefits to using a mobile electronic check register?
2. How does the CheckRegister MIDlet handle showing the balance to the user?
3. Why is it necessary to store the transaction amount in cents?

Exercises

1. One significant weakness with the CheckRegister MIDlet is that it doesn't attempt to keep the transactions in any order. Modify the MIDlet so that the transactions are ordered according to their dates.
2. Based on the previous Q&A question, modify the CheckRegister MIDlet so that it supports multiple accounts (savings, checking, and so forth).

DAY 16

Bidding on the Go

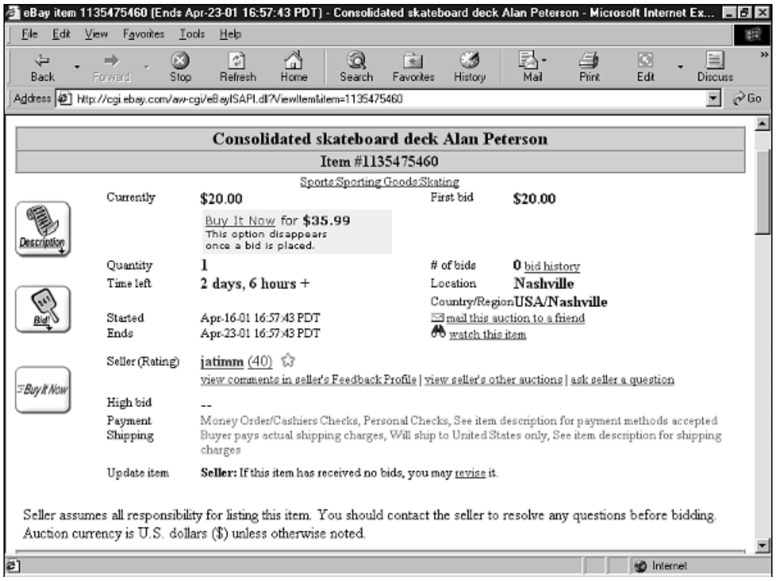
If you're like me, at some point you might have been skeptical about the concept of online auctions. However, as soon as you realized how neat it is to find obscure collectibles and bargains of all kinds in online auctions, you became somewhat of an auction junkie. Perhaps "junkie" is a bit of a strong term, but the reality is that many of people are warming up to the idea of online auctions. What at one time seemed impractical due to the uncertainties of carrying out financial transactions with strangers in distant locations has quickly evolved into a surprisingly reliable global yard sale. Online auctions are clearly here to stay, and they provide an interesting opportunity for mobile device users.

This lesson focuses on the design and development of an auction watch MIDlet that tracks online auctions. The idea behind the MIDlet is to provide a means of entering items that you want to watch, and then being able to keep track of the bid amount and time remaining on the auction while you're out and about with your phone or pager. The following main topics are covered in this lesson:

- Determining the usefulness of tracking online auctions with a mobile device
- Designing and building an auction watch MIDlet
- Testing the auction watch MIDlet

Keeping Up with Online Auctions from Afar

If you've yet to experience the thrill of online auctioning, please stop reading this lesson right now and go bid on something you don't need. There are several different auction Web sites out there including eBay (`http://www.ebay.com/`), Yahoo! Auctions (`http://auctions.yahoo.com/`), and Amazon.com Auctions (`http://auctions.amazon.com/`); eBay is the largest. Online auctioning is a great way to find bargains on anything from sports collectibles to arcade games to automobiles. Yes, even automobiles are now being sold through online auctions! Don't worry, I'm not getting any kickbacks from auction Web sites by encouraging you to try them out. I've just found them to be both valuable and entertaining as I've hunted for my own deals. Figure 16.1 shows an item for auction on the eBay Web site, and Figure 16.2 shows a similar item on Yahoo! Auctions.


eBay item 1135475460 (Ends Apr-23-01 16:57:43 PDT) - Consolidated skateboard deck Alan Peterson - Microsoft Internet Ex...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://cgi.ebay.com/aw-cgi/eBayISAPI.dll?ViewItem&item=1135475460
Go
Description
Bid!
Buy It Now
Internet

Consolidated skateboard deck Alan Peterson

Item #1135475460

Sports:Sporting Goods:Skating

Currently	**$20.00**	First bid	**$20.00**
	Buy It Now for **$35.99** This option disappears once a bid is placed.		
Quantity	**1**	# of bids	**0** bid history
Time left	**2 days, 6 hours +**	Location	**Nashville**
		Country/Region	**USA/Nashville**
Started	Apr-16-01 16:57:43 PDT		mail this auction to a friend
Ends	Apr-23-01 16:57:43 PDT		watch this item
Seller (Rating)	jatimm (40) view comments in seller's Feedback Profile | view seller's other auctions | ask seller a question		
High bid	--		
Payment	Money Order/Cashiers Checks, Personal Checks, See item description for payment methods accepted		
Shipping	Buyer pays actual shipping charges, Will ship to United States only, See item description for shipping charges		
Update item	**Seller:** If this item has received no bids, you may revise it.		

Seller assumes all responsibility for listing this item. You should contact the seller to resolve any questions before bidding. Auction currency is U.S. dollars ($) unless otherwise noted.

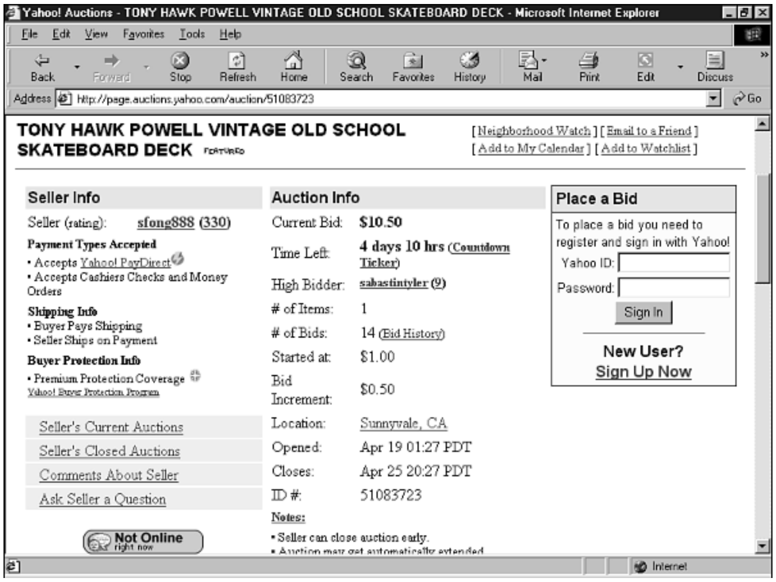
FIGURE 16.1

eBay is without a doubt the largest online auction Web site, and boasts over 5 million items for sale.

Of course, one problem with online auctions is that they usually last several days, which means you must keep track of when they end or you will likely miss out on any last minute bidding. To win an auction and get what you want requires some vigilance. I've missed out on getting several items because I forgot when the auction was ending and had no way of checking the status of it because I was away from my computer. However, my problems can now be solved, thanks to mobile devices, which now make it possible to obtain auction data wherever you are.

FIGURE 16.2

Yahoo! Auctions is the runner-up to eBay, and offers many of the same items but without nearly as much variety.


Yahoo! Auctions - TONY HAWK POWELL VINTAGE OLD SCHOOL SKATEBOARD DECK - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://page.auctions.yahoo.com/auction/51083723
Go
Internet

TONY HAWK POWELL VINTAGE OLD SCHOOL SKATEBOARD DECK FEATURED

[Neighborhood Watch] [Email to a Friend]
[Add to My Calendar] [Add to Watchlist]

Seller Info

Seller (rating): sfong888 (330)

Payment Types Accepted
- Accepts Yahoo! PayDirect
- Accepts Cashiers Checks and Money Orders

Shipping Info
- Buyer Pays Shipping
- Seller Ships on Payment

Buyer Protection Info
- Premium Protection Coverage
Yahoo! Buyer Protection Program

Seller's Current Auctions
Seller's Closed Auctions
Comments About Seller
Ask Seller a Question

Not Online right now

Auction Info

Current Bid:	**$10.50**
Time Left:	**4 days 10 hrs** (Countdown Ticker)
High Bidder:	sabastintyler (9)
# of Items:	1
# of Bids:	14 (Bid History)
Started at:	$1.00
Bid Increment:	$0.50
Location:	Sunnyvale, CA
Opened:	Apr 19 01:27 PDT
Closes:	Apr 25 20:27 PDT
ID #:	51083723

Notes:
- Seller can close auction early.
- Auction may get automatically extended

Place a Bid

To place a bid you need to register and sign in with Yahoo!

Yahoo ID:
Password:
Sign In

New User?
Sign Up Now

Note

It's worth pointing out that eBay currently allows you to access auction information wirelessly using WAP. If you have a WAP phone then it is possible to use a special Web browser on the phone to view eBay auctions. For more information, visit `http://www.ebay.com`.

Of course, keeping track of auction information isn't restricted to the buying side of the equation. You might also want to keep up with items you have for sale while you're away from your desktop computer. As an example, the automotive section of eBay includes many automobiles that are for sale on the lot at car dealerships. These automobiles typically have a reserve on their auction listing, which means that they don't sell unless the bidding gets up to an established minimum amount. After the bidding reaches the reserve, the car dealer is bound to the terms of the auction and must sell the car to the high bidder. Of course, the car is still sitting on the lot at the moment this takes place, which could feasibly cause a problem if someone walks up and wants to buy it. Therefore, it could be beneficial for salespeople at these dealerships to be able to track the status of auctions live from their mobile devices as they are working the showroom floor.

Hopefully, I've convinced you that an auction watch MIDlet could be a neat tool for those of us who have caught the auction bug. Keep in mind that in addition to people who use online auctions to find bargains and hard-to-find collectibles, there are also people who run full-blown businesses that sell solely through online auctions. These people have their entire livelihood tied to online auction Web sites, and therefore have a very serious interest in keeping up with live auction information. An auction watch MIDlet could prove even more valuable to them.

With the case made for building an auction watch MIDlet, let's assess what is required of the MIDlet in terms of data. It's fairly obvious that the logical unit of data in the MIDlet will be an auction item. An auction item on a typical auction Web site has several pieces of information including a unique ID, a description, the current high bid amount, the time remaining, and the name of the high bidder. You might initially think that it would make sense to store all of this information persistently for each auction item that you want to track, but this simply isn't the case. With the bid amount and time remaining changing constantly, it doesn't make sense to store those pieces of information persistently. In fact, the only two pieces of item information that you must store are the ID and the description.

The reason for distilling an auction item down to two simple pieces of information is to allow the MIDlet to focus on retrieving the other item information in real-time so that it is up-to-date. The item description could also be retrieved with the other live data, but in this case you will probably want to enter your own item description because the ones used in most auction listings are too long to be easily displayed in the context of a small mobile device screen.

To enter an item in the auction watch MIDlet, you must enter the ID of the item and a brief description. The description identifies the item in a watch list and the ID retrieves the live bid amount and time remaining from the auction Web site. The information for an auction item breaks down as follows:

- **ID**—Stored persistently
- **Description**—Stored persistently
- **Bid Amount**—Retrieved live
- **Time Remaining**—Retrieved live

You might wonder how to handle each item in the auction watch list having data that is obtained in two different ways. The approach I like involves displaying each item description as it is read from an RMS record store, and then only displaying the live auction information if the user requests it. In this way you minimize network delays and focus solely on updating items in the list on an individual basis.

That wraps up the basic design of the auction watch MIDlet. You're now ready to dig into the code and see how the MIDlet comes together.

Constructing the AuctionWatch MIDlet

We saw that it would be necessary to use the RMS to store item information persistently. However, this information is limited to the item ID and description, and not all of the information associated with an auction item. Considering that you've now worked through the development of three MIDlets that utilized the RMS for persistent data storage, the RMS aspects of the AuctionWatch MIDlet should be pretty familiar. The AuctionWatch MIDlet is unique among the other RMS MIDlets in that it combines persistent data with real-time data when managing auction items. This is an interesting merger of two data management approaches that happens to work out great for the AuctionWatch MIDlet.

The construction of the AuctionWatch MIDlet follows a series of steps similar to those you've followed when developing the other RMS MIDlets. Even so, you'll find the AuctionWatch MIDlet an interesting departure in some ways from the other information-based MIDlets because of how it ties into live auction data.

Managing the Item Records

As you probably could guess, the AuctionWatch MIDlet relies on a couple of helper classes to represent an individual auction item and the item record store. The two classes that support the management of the persistent storage of auction items are

- **`Item`**—Represents a single item in the auction watch list and record store
- **`ItemDB`**—Encapsulates the record store that stores item records

The next two sections show you how these classes work. You'll find that the `Item` class is relatively simple because of limited data requirements of auction items, whereas the `ItemDB` class is very familiar to the record store classes you've seen in other RMS MIDlets.

The `Item` Class

The `Item` class is responsible for providing a means of accessing the persistent information associated with an auction item. More specifically, the `Item` class includes member variables that store the ID and description of an auction item, as well as methods that make it possible to get the values of these member variables. The most important functionality provided by the `Item` class consists of the methods that pack and unpack item data for persistent storage. The complete code for the `Item` class follows in Listing 16.1.

LISTING 16.1 The `Item` Class Represents a Single Auction Item

```
import java.util.*;

public class Item {
  private int number;
  private String description;

  public Item(int n, String d) {
    number = n;
    description = d;
  }

  public Item(byte[] data) {
    unpack(new String(data));
  }

  public void unpack(String data) {
    int start = 0, end = data.indexOf(';');
    number = Integer.parseInt(data.substring(start, end));
    start = end + 1;
    description = data.substring(start, data.length());
  }

  public String pack() {
    return (String.valueOf(number) + ';' + description);
  }

  public int getNumber() {
    return number;
  }

  public String getDescription() {
    return description;
  }
}
```

As this code reveals, the `Item` class is relatively simple. The `number` and `description` member variables store the two pieces of information associated with an auction item. The member variables are initialized in the two `Item()` constructors. The first constructor accepts two values that map to each of the member variables, and the second constructor creates an `Item` object from a byte array of data. The second `Item()` constructor is extremely important because it is used when you read an item from the item record store. This constructor calls the `unpack()` method, which handles the details of parsing through the packed item data and carefully extracting the two pieces of item information. Of course, you also need a `pack()` method that packs together item data for storage; the `pack()` method takes the two pieces of item information and packs them into a semicolon-delimited string. Together, the `pack()` and `unpack()` methods provide the interface required to store and retrieve auction items to and from the item record store.

The remaining `getNumber()` and `getDescription()` methods in the `Item` class are access methods that are used to retrieve the individual pieces of item information. These methods are used in the main `AuctionWatch` MIDlet class whenever it is necessary to obtain an item number or description from an `Item` object. These methods also serve to complete the code for the `Item` class.

The `ItemDB` Class

The `ItemDB` class is used to encapsulate the record store for auction items. In many ways the wrapper class for the record store is similar in all RMS MIDlets. In fact, the `ItemDB` class is virtually identical to the `TransactionDB` class from the preceding lesson. The only noticeable difference between the two classes is that the class types are different given that one operates on `Item` objects and the other operates on `Transaction` objects. You could actually generalize the record store class and have it work with the `Object` class instead of specific derived classes such as `Item` and `Transaction`. This would enable you to share the code between different RMS MIDlets. However, for the purposes of this book it is a little cleaner to create different record store classes that use the exact types required of each MIDlet.

The complete source code for the `ItemDB` class is in Listing 16.2.

LISTING 16.2 The `ItemDB` Class Manages a Record Store of Auction Items

```
import javax.microedition.rms.*;
import java.util.Enumeration;
import java.util.Vector;
import java.io.*;

public class ItemDB {
  RecordStore recordStore = null;

  public ItemDB(String name) {
    // Open the record store using the specified name
    try {
      recordStore = open(name);
    }
    catch(RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public RecordStore open(String fileName) throws RecordStoreException {
    return RecordStore.openRecordStore(fileName, true);
  }

  public void close() throws RecordStoreNotOpenException,
    RecordStoreException {
```

LISTING 16.2 continued

```
    // If the record store is empty, delete the file
    if (recordStore.getNumRecords() == 0) {
      String fileName = recordStore.getName();
      recordStore.closeRecordStore();
      recordStore.deleteRecordStore(fileName);
    }
    else {
      // Otherwise, close the record store
      recordStore.closeRecordStore();
    }
  }

  public Item getItemRecord(int id) {
    // Get the item record from the record store
    try {
      return (new Item(recordStore.getRecord(id)));
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }

    return null;
  }

  public synchronized void setItemRecord(int id, String record) {
    // Convert the string record to an array of bytes
    byte[] bytes = record.getBytes();

    // Set the record in the record store
    try {
      recordStore.setRecord(id, bytes, 0, bytes.length);
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public synchronized int addItemRecord(String record) {
    // Convert the string record to an array of bytes
    byte[] bytes = record.getBytes();

    // Add the byte array to the record store
    try {
      return recordStore.addRecord(bytes, 0, bytes.length);
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }
```

LISTING 16.2 continued

```
    return -1;
  }

  public synchronized void deleteItemRecord(int id) {
    // Delete the item record from the record store
    try {
      recordStore.deleteRecord(id);
    }
    catch (RecordStoreException e) {
      e.printStackTrace();
    }
  }

  public synchronized RecordEnumeration enumerateItemRecords()
    throws RecordStoreNotOpenException {
    return recordStore.enumerateRecords(null, null, false);
  }
}
```

This class is very similar to record store wrapper classes used in the other RMS MIDlets, so you should be familiar with how it works. To recap, the following is a list of the methods in the `ItemDB` class, along with what they do:

- **`open()`**—Opens the item record store
- **`close()`**—Closes the item record store
- **`getItemRecord()`**—Retrieves a item record from the record store
- **`setItemRecord()`**—Alters an existing item record in the record store by setting it with new item data
- **`addItemRecord()`**—Adds a new item record to the record store
- **`deleteItemRecord()`**—Deletes a item record from the record store
- **`enumerateItemRecords()`**—Obtains an enumeration of the item records in the record store

These methods perform standard RMS record store operations and enable the AuctionWatch MIDlet to store, retrieve, and enumerate the item record store with ease. You're now ready to see how the `AuctionWatch` MIDlet class is put together.

The User Interface

As you know from earlier in the lesson, the AuctionWatch MIDlet involves the user entering auction items and then retrieving live data for those items. The AuctionWatch MIDlet has three primary functions: interacting with the user through its user interface, storing and retrieving item data from and to a persistent record store, and obtaining live item data from an auction Web site. We will begin the assembly of the MIDlet by focusing on the user interface because the other features flow from user interface commands.

The user interface for the AuctionWatch MIDlet consists of two different screens and a few other GUI components. The main screen of the MIDlet is responsible for listing all of the items that have been stored in the record store. Of course, there must be some way to enter new items, so another screen is required for entering the information associated with a new item. This information consists of the item ID and description, which can both be entered into text fields. The first place to start in constructing the user interface for a MIDlet is the MIDlet class's member variables.

The member variables for the AuctionWatch MIDlet are listed next. They are used to store the GUI components of the MIDlet, along with a few other pieces of the MIDlet puzzle.

```
private Command exitCommand, addCommand, deleteCommand, backCommand,
  saveCommand;
private Display display;
private List mainScreen;
private Form itemScreen;
private TextField numberField, descriptionField;
private ItemDB db = null;
private Vector itemIDs = new Vector();
```

Not surprisingly, the commands used by the MIDlet appear first in the list of member variables. The `Exit`, `Add`, and `Delete` commands appear on the main screen, and the `Back` and `Save` commands appear on the item form screen. Selecting the `Add` command displays the item form screen; this form enables the user to enter a new item. The new item can then be added to the record by selecting the `Save` command, or cancelled by using the `Back` command. To delete an item, select the `Delete` command from the main screen. The most important feature of the MIDlet is the retrieval of live auction information, which is carried out when you use the Select button on the device. When you press the Select button, the live auction information for the currently selected auction item is obtained and added to the item entry in the auction list.

Along with the commands, member variables for the AuctionWatch MIDlet include the standard `Display` object, a `List` object for the main screen, and a `Form` object for the item screen. Two text fields are then created to accommodate the entry of the two pieces of item information when adding an item to the item list.

The `ItemDB` record store object is declared next, along with `itemIDs`, which is a vector that keeps track of the record ID for each item in the item list. You remember from other RMS MIDlets that this vector is necessary so that the item list can remain synchronized with the record store.

The member variables for the AuctionWatch MIDlet are initialized in the `AuctionWatch()` constructor, whose code follows in Listing 16.3.

LISTING 16.3 The `AuctionWatch()` Constructor Is Responsible for Initializing the MIDlet

```
public AuctionWatch() {
  // Get the Display object for the MIDlet
  display = Display.getDisplay(this);

  // Create the commands
  exitCommand = new Command("Exit", Command.EXIT, 2);
  addCommand = new Command("Add", Command.SCREEN, 3);
  deleteCommand = new Command("Delete", Command.SCREEN, 3);
  backCommand = new Command("Back", Command.BACK, 2);
  saveCommand = new Command("Save", Command.OK, 3);

  // Create the main screen
  mainScreen = new List("Items", List.IMPLICIT);

  // Set the Exit and Add commands for the main screen
  mainScreen.addCommand(exitCommand);
  mainScreen.addCommand(addCommand);
  mainScreen.addCommand(deleteCommand);
  mainScreen.setCommandListener(this);

  // Create the item screen
  itemScreen = new Form("Item Info");
  numberField = new TextField("Item #", "", 12, TextField.NUMERIC);
  itemScreen.append(numberField);
  descriptionField = new TextField("Description", "", 30, TextField.ANY);
  itemScreen.append(descriptionField);

  // Set the Back and Save commands for the item screen
  itemScreen.addCommand(backCommand);
  itemScreen.addCommand(saveCommand);
  itemScreen.setCommandListener(this);

  // Open the item database
  try {
    db = new ItemDB("items");
  }
  catch(Exception e) {
    System.err.println("EXCEPTION: Problem opening the database.");
  }

  // Read through the database and build a list of record IDs
  RecordEnumeration records = null;
  try {
    records = db.enumerateItemRecords();
```

LISTING 16.3 continued

```
      while(records.hasNextElement())
        itemIDs.addElement(new Integer(records.nextRecordId()));
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the item records.");
    }

    // Read through the database and fill the item list
    records.reset();
    try {
      while(records.hasNextElement()) {
        Item item = new Item(records.nextRecord());
        mainScreen.append(item.getDescription(), null);
      }
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the item records.");
    }
  }
```

The `AuctionWatch()` constructor not only initializes the member variables, but it also carries out quite a bit of setup work for the MIDlet. The constructor begins by obtaining the `Display` object for the MIDlet and storing it in the `display` member variable. With that taken care of, the commands are then created with their appropriate priority levels. The main screen is created next, followed by the item screen, and commands are added to each.

With the user interface components set up and ready to go, the `AuctionWatch()` constructor turns its attention to the item record store. The constructor first reads through the record store to construct the vector of record IDs. When that is complete, the constructor resets the record store enumeration and steps through the records again. On the second pass through the record store each item is actually added to the item list. To do this, a temporary `Item` object must be created for each record in the record store, after which the `getDescription()` method is used to add the item to the list.

Handling Commands

As you might have guessed, the other big chunk of code in the AuctionWatch MIDlet is the `commandAction()` method, which is responsible for handling commands issued by the user. If you recall, the `Exit`, `Add`, `Delete`, and `Select` commands apply to the main list screen, while the `Back` and `Save` commands are accessible from the item form screen. The complete code for the `commandAction()` method is shown in Listing 16.4. It shows exactly how each of the MIDlet commands are carried out.

LISTING 16.4 The `commandAction()` Method Handles Commands for the MIDlet

```
public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == addCommand) {
    // Clear the item fields
    numberField.setString("");
    descriptionField.setString("");

    // Set the current display to the item screen
    display.setCurrent(itemScreen);
  }
  else if (c == deleteCommand) {
    // Get the record ID of the currently selected item
    int index = mainScreen.getSelectedIndex();
    int id = ((Integer)itemIDs.elementAt(index)).intValue();

    // Delete the item record
    db.deleteItemRecord(id);
    itemIDs.removeElementAt(index);
    mainScreen.delete(index);
  }
  else if (c == List.SELECT_COMMAND) {
    // Get the record ID of the currently selected item
    int index = mainScreen.getSelectedIndex();
    int id = ((Integer)itemIDs.elementAt(index)).intValue();

    // Retrieve the item record from the database
    Item item = db.getItemRecord(id);

    // Get the live item information
    String liveItemInfo = getLiveItemInfo(item.getNumber());

    // Update the item in the item list with the live info
    mainScreen.set(index, item.getDescription() + "\n" + liveItemInfo, null);
  }
  else if (c == backCommand) {
    // Set the current display back to the main screen
    display.setCurrent(mainScreen);
  }
  else if (c == saveCommand) {
    // Create a record for the item
    int itemNum = 0;
    if (numberField.getString().length() > 0)
      itemNum = Integer.parseInt(numberField.getString());
    Item item = new Item(itemNum, descriptionField.getString());
```

LISTING 16.4 continued

```
      // Add the item to the database
      itemIDs.addElement(new Integer(db.addItemRecord(item.pack())));
      mainScreen.append(item.getDescription(), null);

      // Set the current display back to the main screen
      display.setCurrent(mainScreen);
    }
  }
```

The `commandAction()` method starts out with the `Exit` command, which I'm sure you recognize as it relies on standard exit code used in all of the MIDlets in this book. The `Add` command is tackled next, and involves clearing the two text fields on the item form and setting the current display to the item screen. The `Delete` command is then handled, which involves deleting the currently selected item from the item list and the record store. The `Select` command is the next command handled by the `commandAction()` method. This command code obtains the ID of the currently selected item, and then uses it to obtain the number of the item. If you recall from earlier in the lesson, this item number is the unique number that identifies the item on an auction Web site. The item number is passed into the `getLiveItemInfo()` method, which obtains a string containing the current bid amount and time remaining on the item in the auction. You learn about this method in just a moment. The code for the `Select` command finishes by updating the item in the item list to reflect the live information.

The `Back` command code is associated with the item screen, and enables the user to cancel out of adding a net item. The code for this handler is very simple in that it only has to set the current display back to the main screen. The other command associated with the item screen is `Save`, which saves the newly added item to the record and adds it to the main item list. After saving the item, the `Save` command code finishes by setting the current display back to the main list screen.

Retrieving the Live Auction Data

Aside from the RMS code involved in saving and retrieving item records to and from the item record store, the other significant piece of functionality in the AuctionWatch MIDlet is the code that retrieves live item information from an auction Web site. Up until this point in the construction of the MIDlet I haven't mentioned any specifics about the auction site used to obtain live item information. That is because the majority of the code in the AuctionWatch MIDlet could be applied to any online auction. However, now it is necessary to commit to a particular site because you will be parse through auction data to extract the current bid amount and remaining time on an item.

With the AuctionWatch MIDlet you can track auctions listed on the Yahoo! Auctions Web site, shown in Figure 16.2. The MIDlet relies on the `getLiveItemInfo()` method to extract live item information from a Web page that is dynamically generated by the Yahoo! Auctions Web site. To accomplish this, the `getLiveItemInfo()` method must create a URL for the item Web page, open a network connection, read the page, and parse out the live item information. Before you jump into the code for this method, take a closer look at exactly what is required of the `getLiveItemInfo()` method:

1. Construct a URL for the auction item Web page.
2. Open a network connection to the URL.
3. Read data from the network connection and search for the pieces of live auction item information.
4. When the item information is found, format it into a simple string.

These steps are still somewhat general, but they give you a good idea as to what the `getLiveItemInfo()` method must accomplish to successfully obtain the live item information from the Yahoo! Auctions Web site. However, it is still important to understand how an item page on the auction site is structured. First, you must understand how to assemble a URL for an item Web page given the item number. Fortunately, this is quite simple. The URL for an auction item Web page is as follows:

```
http://page.auctions.yahoo.com/auction/51083723
```

The end of the URL is the number of the item, while the remaining portion of the URL is fixed for all auctions. So you can easily obtain the Web page for any auction item by appending the item number to the string `"http://page.auctions.yahoo.com/auction/"`.

Now that you understand how to obtain the Web page for an auction, let's take a look at what such a page actually looks like in HTML code. The following is an excerpt of the HTML code for an item Web page on the Yahoo! Auctions Web site:

```
<TR><TD colspan=2 BGCOLOR="#ffe566"><B><FONT FACE=arial>Auction Info</FONT>
</B></TD></TR>
<TR><TD>Current Bid:</TD><TD><b>$13.50</b></TD></TR>
<TR><TD>Time Left:</TD><TD><b>4&nbsp;days&nbsp;1&nbsp;hr</b> <small>
(<a href="javascript:var junk=window.open(
'http://page.auctions.yahoo.com/show/countdown?aID=51083723',
'countdown51083723', 'width=400,height=190,scrollbars=yes,
resizable=yes,status=0')"><b>Countdown Ticker</b></a>)</small></TD></TR>
```

This code is pulled from the middle of the Web page but it nonetheless highlights the relevant information that you must extract for use in the AuctionWatch MIDlet. The third line of code shows the current bid amount, which in this case is `$13.50`. If you move down to the next line you'll see the time remaining on the auction. The time remaining is a little tougher to make out because it has the special `&nbsp;` HTML symbol between the numbers and words. However, if you visualize spaces in place of the `&nbsp;` symbols you'll quickly realize that the time remaining for this auction item is `4 days 1 hr`.

The task for the `getLiveItemInfo()` method is to read through the auction item Web page and extract the bid amount and time remaining. Although there are certainly more robust solutions that you could use, one relatively easy way to find the bid amount is to look for the first occurrence of a dollar sign (`$`). When you find a dollar sign, you know you've arrived at the bid amount because it is the first monetary value listed in a Yahoo! Auctions Web page. Then you can read the bid amount by reading the characters after the dollar sign until you reach a less-than symbol (<). Take a look at the preceding code listing to see why this works.

After reading the bid amount for the item, you can skip forward a fixed number of spaces to extract the time remaining. This works because the same HTML tags are always used to separate the bid amount and the time remaining in all Yahoo! Auctions item Web pages. When you get to the time remaining, you can read it by grabbing characters until you reach another less-than symbol (<). Finally, you'll need to replace those pesky `&nbsp;` symbols with spaces.

This technique of extracting the item information might sound a little tricky, but it's really not too bad. Check out the code for the `getLiveItemInfo()` method in Listing 16.5 to see what I mean.

LISTING 16.5 The `getLiveItemInfo()` Method Retrieves the Live Information for an Auction from the Yahoo! Auctions Web Site

```
private String getLiveItemInfo(int inum) {
  StreamConnection conn = null;
  InputStream in = null;
  StringBuffer bidAmount = new StringBuffer(),
    time = new StringBuffer();

  // Build the URL for the Yahoo auction item page
  String url = "http://page.auctions.yahoo.com/auction/" + String.valueOf(inum);

  try {
    // Open the HTTP connection
    conn = (StreamConnection)Connector.open(url);

    // Obtain an input stream for the connection
    in = conn.openInputStream();

    // Read through the page, finding the bid amount and time remaining
```

LISTING 16.5 continued

```
      int ch;
      boolean readingBidAmount = false, countingToTime = false,
        readingTime = false, finished = false;
      int timeCountdown = 44;
      while ((ch = in.read()) != -1 && !finished) {
        // Find the first dollar sign
        if (((char)ch) == '$')
          readingBidAmount = true;

        if (readingBidAmount) {
          // Read the bid amount
          if (((char)ch) != '<')
            bidAmount.append((char)ch);
          else {
            readingBidAmount = false;
            countingToTime = true;
          }
        }
        else if (countingToTime) {
          // Skip over to the time remaining
          if (--timeCountdown == 0) {
            countingToTime = false;
            readingTime = true;
          }
        }
        else if (readingTime) {
          // Read the time remaining
          if (((char)ch) != '<')
            time.append((char)ch);
          else
            finished = true;
        }
      }
    }
    catch (IOException e) {
      System.err.println("The connection could not be established.");
    }

    // Remove all "&nbsp;" occurrences from the time string
    int i = 0;
    char c;
    while (i < time.length()) {
      c = time.charAt(i);
      if (c == '&') {
        time.delete(i, i + 6);
        time.insert(i, ' ');
      }
      i++;
```

LISTING 16.5 continued

```
    }

    // Display an error message if necessary
    if (bidAmount.length() == 0)
      display.setCurrent(new Alert("Auction Watch",
        "The item is invalid. Please check its item number.", null,
➥AlertType.ERROR));

    return (bidAmount + " " + time);
  }
```

The `getLiveItemInfo()` method accepts an integer item number as its only parameter, which it uses to construct a URL for the item Web page. The method then uses the `Connector` class to establish a network connection to the URL. This connection is cast to a `StreamConnection` object, which is then used as the basis for opening an input stream. The interesting code in the `getLiveItemInfo()` method is located within the `while` loop that reads a character at a time from the input stream. Notice that just before the `while` loop executes, several local variables are created and initialized; these local variables establish the different tasks that are carried out while reading through the Web page data.

Within the `while` loop, each character is tested to see whether it is a dollar sign (`$`). If a match is found, the `readingBidAmount` variable is set to `true`, and the method begins reading the bid amount. While the bid amount is being read, each character is checked to see if it is a less-than character (<). If so, you know you've reached the end of the bid amount; if not, the method continues to read more characters.

After reading the bid amount, the `getLiveItemInfo()` method begins skipping characters using the `countingToTime` variable. After the characters are skipped over, the next data available will be the time remaining. So, the `readingTime` variable is set to `true` and the code begins reading the time from the Web data. After reading the time remaining, the `finished` flag is set to `true`, which causes the `while` loop to exit. Keep in mind that the time variable still contains a string with the `&nbsp;` symbols, which must be converted to spaces. The method tackles that problem next, which results in a `time` variable that is ready to be returned to the MIDlet along with the bid amount.

Just before returning the item information, the `getLiveItemInfo()` method checks to make sure that the `bidAmount` variable contains data; if it doesn't, an error message is displayed. The method then builds a string containing both the bid amount and the time remaining, and returns it to the MIDlet.

Putting It All Together

At long last, the different components of the AuctionWatch MIDlet have now been designed, built, and are ready to go. Let's revisit the entire code for the MIDlet class so you can see how it all fits together (see Listing 16.6.).

LISTING 16.6 The Complete Source Code for the `AuctionWatch` Class

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;

public class AuctionWatch extends MIDlet implements CommandListener {
  private Command exitCommand, addCommand, deleteCommand, backCommand,
    saveCommand;
  private Display display;
  private List mainScreen;
  private Form itemScreen;
  private TextField numberField, descriptionField;
  private ItemDB db = null;
  private Vector itemIDs = new Vector();

  public AuctionWatch() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    addCommand = new Command("Add", Command.SCREEN, 3);
    deleteCommand = new Command("Delete", Command.SCREEN, 3);
    backCommand = new Command("Back", Command.BACK, 2);
    saveCommand = new Command("Save", Command.OK, 3);

    // Create the main screen
    mainScreen = new List("Items", List.IMPLICIT);

    // Set the Exit and Add commands for the main screen
    mainScreen.addCommand(exitCommand);
    mainScreen.addCommand(addCommand);
    mainScreen.addCommand(deleteCommand);
    mainScreen.setCommandListener(this);

    // Create the item screen
    itemScreen = new Form("Item Info");
    numberField = new TextField("Item #", "", 12, TextField.NUMERIC);
    itemScreen.append(numberField);
```

LISTING 16.6 continued

```
    descriptionField = new TextField("Description", "", 30, TextField.ANY);
    itemScreen.append(descriptionField);

    // Set the Back and Save commands for the item screen
    itemScreen.addCommand(backCommand);
    itemScreen.addCommand(saveCommand);
    itemScreen.setCommandListener(this);

    // Open the item database
    try {
      db = new ItemDB("items");
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem opening the database.");
    }

    // Read through the database and build a list of record IDs
    RecordEnumeration records = null;
    try {
      records = db.enumerateItemRecords();
      while(records.hasNextElement())
        itemIDs.addElement(new Integer(records.nextRecordId()));
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the item records.");
    }

    // Read through the database and fill the item list
    records.reset();
    try {
      while(records.hasNextElement()) {
        Item item = new Item(records.nextRecord());
        mainScreen.append(item.getDescription(), null);
      }
    }
    catch(Exception e) {
      System.err.println("EXCEPTION: Problem reading the item records.");
    }
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the main screen
    display.setCurrent(mainScreen);
  }

  public void pauseApp() {
  }
```

Listing 16.6 continued

```
public void destroyApp(boolean unconditional) {
  // Close the item database
  try {
    db.close();
  }
  catch(Exception e) {
    System.err.println("EXCEPTION: Problem closing the database.");
  }
}

public void commandAction(Command c, Displayable s) {
  if (c == exitCommand) {
    destroyApp(false);
    notifyDestroyed();
  }
  else if (c == addCommand) {
    // Clear the item fields
    numberField.setString("");
    descriptionField.setString("");

    // Set the current display to the item screen
    display.setCurrent(itemScreen);
  }
  else if (c == deleteCommand) {
    // Get the record ID of the currently selected item
    int index = mainScreen.getSelectedIndex();
    int id = ((Integer)itemIDs.elementAt(index)).intValue();

    // Delete the item record
    db.deleteItemRecord(id);
    itemIDs.removeElementAt(index);
    mainScreen.delete(index);
  }
  else if (c == List.SELECT_COMMAND) {
    // Get the record ID of the currently selected item
    int index = mainScreen.getSelectedIndex();
    int id = ((Integer)itemIDs.elementAt(index)).intValue();

    // Retrieve the item record from the database
    Item item = db.getItemRecord(id);

    // Get the live item information
    String liveItemInfo = getLiveItemInfo(item.getNumber());

    // Update the item in the item list with the live info
    mainScreen.set(index, item.getDescription() + "\n" + liveItemInfo, null);
  }
  else if (c == backCommand) {
```

LISTING 16.6 continued

```
        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
      else if (c == saveCommand) {
        // Create a record for the item
        int itemNum = 0;
        if (numberField.getString().length() > 0)
          itemNum = Integer.parseInt(numberField.getString());
        Item item = new Item(itemNum, descriptionField.getString());

        // Add the item to the database
        itemIDs.addElement(new Integer(db.addItemRecord(item.pack())));
        mainScreen.append(item.getDescription(), null);

        // Set the current display back to the main screen
        display.setCurrent(mainScreen);
      }
    }

    private String getLiveItemInfo(int inum) {
      StreamConnection conn = null;
      InputStream in = null;
      StringBuffer bidAmount = new StringBuffer(),
        time = new StringBuffer();

      // Build the URL for the Yahoo auction item page
      String url = "http://page.auctions.yahoo.com/auction/" +
  String.valueOf(inum);

      try {
        // Open the HTTP connection
        conn = (StreamConnection)Connector.open(url);

        // Obtain an input stream for the connection
        in = conn.openInputStream();

        // Read through the page, finding the bid amount and time remaining
        int ch;
        boolean readingBidAmount = false, countingToTime = false,
          readingTime = false, finished = false;
        int timeCountdown = 44;
        while ((ch = in.read()) != -1 && !finished) {
          // Find the first dollar sign
          if (((char)ch) == '$')
            readingBidAmount = true;

          if (readingBidAmount) {
            // Read the bid amount
```

LISTING 16.6 continued

```
          if (((char)ch) != '<')
            bidAmount.append((char)ch);
          else {
            readingBidAmount = false;
            countingToTime = true;
          }
        }
        else if (countingToTime) {
          // Skip over to the time remaining
          if (--timeCountdown == 0) {
            countingToTime = false;
            readingTime = true;
          }
        }
        else if (readingTime) {
          // Read the time remaining
          if (((char)ch) != '<')
            time.append((char)ch);
          else
            finished = true;
        }
      }
    }
    catch (IOException e) {
      System.err.println("The connection could not be established.");
    }

    // Remove all "&nbsp;" occurrences from the time string
    int i = 0;
    char c;
    while (i < time.length()) {
      c = time.charAt(i);
      if (c == '&') {
        time.delete(i, i + 6);
        time.insert(i, ' ');
      }
      i++;
    }

    // Display an error message if necessary
    if (bidAmount.length() == 0)
      display.setCurrent(new Alert("Auction Watch",
        "The item is invalid. Please check its item number.",
➥null, AlertType.ERROR));

    return (bidAmount + " " + time);
  }
}
```

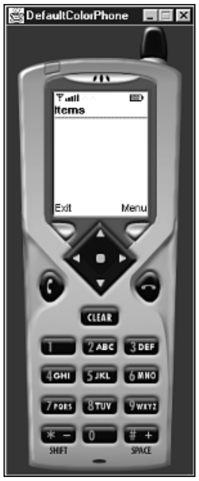
As you can see, the code for this MIDlet is fairly long but not too bad considering that it is using the RMS to store watch items, and then using MIDP I/O features to pull down live auction information from the Yahoo! Auctions Web site. Now you are ready to see the MIDlet in action, so let's try it out in the J2ME emulator.

Testing the AuctionWatch MIDlet

The AuctionWatch MIDlet is an excellent example of how you can provide wireless access to information that is typically difficult to access away from a computer with a full-blown Web browser and a wired Internet connection. However, even with the limited capabilities of MIDP devices, the AuctionWatch MIDlet offers unique functionality that is quite powerful in its simplicity. Figure 16.3 shows the AuctionWatch MIDlet upon first being executed in the J2ME emulator.

FIGURE 16.3

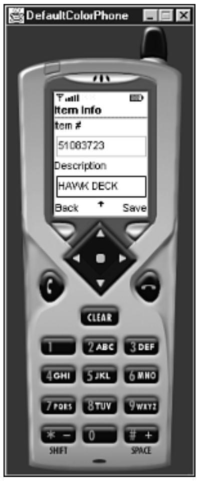
The AuctionWatch MIDlet begins with an empty item list.



The AuctionWatch Midlet starts out with an empty item list, so the first step you must take is to add an auction item that you want to watch. You will need to actually visit the Yahoo! Auctions Web site and find an item that you're interested in watching. You will normally pick several items that you want to be able to track while you're away from your computer. To add a new item, select the `Menu` command followed by the `Add` command. Figure 16.4 shows the Item Info screen, where you enter a new item.

FIGURE 16.4

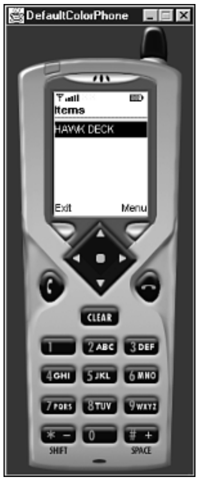
The Item Info screen consists of two input fields that allow you to enter the item number and description for an auction item.



After entering the item information, select the `Save` command to save the item. The description of the item then appears in the main item list, as shown in Figure 16.5.

FIGURE 16.5

After saving a new item, it appears in the main item list.



Now you're ready to grab the live item information by simply selecting the item in the list and then pressing the Select button on the device. After a few moments the list entry for the item will change to reflect the live item information, including the current bid amount and the time remaining on the auction (Figure 16.6).

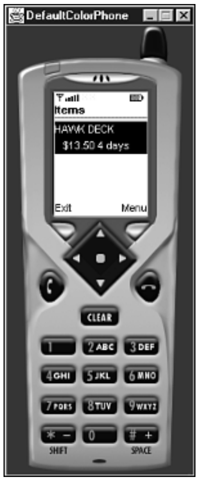


FIGURE 16.6
After pressing the Select button, the item entry in the main item list is updated to reflect the live item information.

Keep in mind that you can delete items by selecting the `Menu` command followed by the `Delete` command. You can also add as many items as you want to track wirelessly. Regardless of how many items you add to the list, you must update the live information for them individually by selecting each item and pressing the Select button.

Summary

Whether or not you're an online auction junkie, you hopefully can see the value of being able to track items up for auction from the convenience of a mobile device. If you buy or sell very often using an online auction site, there are bound to be situations where you'd like to check up on an auction when you're away from your computer. What better place to do so than from your wireless mobile device? This lesson guided you through the design and development of an auction watch MIDlet that gives you to freedom to keep tabs on auction items using a MIDP device.

In this lesson you developed the last MIDlet in a series of highly information-based MIDlets. The next part of the book shifts gears by introducing you to MIDlet animation and games. The next lesson teaches you the basics of MIDlet animation and leads you through the development of a powerful set of reusable MIDlet animation classes.

Q&A

Q Does the AuctionWatch MIDlet work with eBay auctions?

A No. Because the code that parses out the bid amount and time remaining is highly tied to the structure of Yahoo! Auction pages, the MIDlet cannot be used to view the status of eBay auctions. However, a similar approach can be used to obtain eBay auction information.

Q Why aren't the bid amount and time remaining stored in the record store with an auction item?

A These pieces of information are very time-critical, which means that they are constantly changing. Therefore, it makes sense to retrieve and display them on demand, as opposed to storing them away. This helps to ensure that the information is current.

Q Is there a problem if an auction item has a dollar symbol ($) in the title?

A Yes. Because the MIDlet makes the assumption that the first occurrence of a dollar symbol immediately precedes the current bid amount, there is definitely a problem if a dollar symbol appears in the auction title. The only solution is to rethink the manner in which the MIDlet parses out the bid amount from the auction item Web page. For the purposes of this lesson, using the first occurrence of the dollar symbol keeps the code somewhat simpler than a more elaborate (and more stable) approach to parsing out the bid amount.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. How does an online auction differ from a traditional "offline" auction?
2. What is the purpose of an item ID?
3. What two pieces of item information are stored away persistently for the AuctionWatch MIDlet?

Exercises

1. Modify the AuctionWatch MIDlet so that it also displays the high bidder for an auction item in addition to the bid amount and time remaining.
2. Modify the AuctionWatch MIDlet so that it also supports eBay auctions. Hint: You could add a member variable to the `Item` class that indicates whether the item is listed on Yahoo! or eBay. Also, you'll need to figure out how to parse the bid amount and time remaining from eBay auction item pages.

PART IV

Entertainment without the Wires

17 Creating Animated MIDlets

18 An Ode to Pong: Creating MIDlet Games

19 The Next Level of MIDlet Gaming

DAY 17

Creating Animated MIDlets

Because MIDP devices have both graphics capabilities and user input controls, at some point you must consider the possibilities for MIDlet gaming. In many situations a MIDlet game could serve as a useful diversion, so it isn't surprising to expect games to be a significant area of MIDlet development. To build an action game, you must have a means of carrying out at least some level of animation. Without animation, there would be no movement, and without movement there would be no action games. Today's lesson presents the fundamental concepts surrounding MIDlet animation and, more specifically, sprite animation.

NEW TERM *sprite animation*—A form of animation that simulates movement using graphical objects (sprites) that move independently of a background.

After learning the basics of animation and how it applies to MIDlets, you will dig into building a set of sprite animation classes that are powerful and extensive enough to handle all of your MIDlet animation needs. You'll reuse the sprite classes in the next lesson to develop a full-blown action game. These sprite classes handle all the details of managing multiple animated sprite objects with support for transparency, Z-order, collision detection, and custom actions. Don't worry if you aren't familiar with these animation buzzwords because you soon will understand exactly how they impact MIDlet animation. The following topics are covered in this lesson:

- Learning the basics of animation
- Assessing the different types of animation
- Constructing a set of reusable sprite animation classes
- Developing an example MIDlet that puts the sprite classes to use

What Is Animation?

Before seeing animation as it relates to MIDlets, it is important to understand the basics of what animation is and how it works. What is animation? Put simply, *animation* is the illusion of movement. Does this mean all animations you've ever seen are really only illusions? That's exactly right! Perhaps the most surprising animated illusion is one that captured our attention long before computers—the television. When you watch television, you see lots of things moving around. However, what you perceive as movement is really just a trick being played on your eyes.

NEW TERM *animation*—The illusion of movement.

The illusion of movement on television is created by displaying a rapid succession of images with slight changes in content. The human eye perceives these changes as movement because of its low visual acuity, which means that your eyes are fairly easy to trick into believing the illusion of animation. More specifically, the human eye can be tricked into perceiving animated movement with as low as 12 frames of movement per second. It should come as no surprise that this animation speed is the minimum target speed for most computer games. Animation speed is measured in *frames per second (fps)*.

NEW TERM *frames per second (fps)*—The number of animation frames, or image changes, presented every second.

Although 12fps is technically enough to fool your eyes into seeing animation, animations at speeds this low often end up looking somewhat jerky. Therefore, most professional animations use a higher frame rate. Television, for example, uses 30fps. When you go to the movies, you see motion pictures at about 24fps. It is apparent that these frame rates are more than enough to captivate your attention and successfully create the illusion of movement.

When developing animated MIDlets, you typically have the ability to manipulate the frame rate a reasonable amount. The most obvious limitation on frame rate is the speed at which a device can generate and display the animation frames. Because mobile devices have limited memory and processing power, it is not necessarily realistic to try to match television or even desktop computer frame rates in MIDlets. When determining the frame rate for a MIDlet, you usually have some give and take in establishing a low enough frame rate to yield a smooth animation while not bogging down a device's processor and slowing the system down.

Types of Animation

Although the focus of today's lesson is ultimately on sprite animation, it is important to understand the primary types of animation used in animation programming. Actually, many different types of animation exist, all of which are useful in different instances. However, for the purposes of implementing Java animation in MIDlets, I've broken animation down into two basic types: frame-based animation and cast-based animation.

Frame-based Animation

The most simple animation technique is frame-based animation, which finds a lot of usage in animations other than games such as animated presentations. *Frame-based animation* involves simulating movement by displaying a sequence of pregenerated, static frame images. A movie is a perfect example of frame-based animation: Each frame of the film is a frame of animation, and when the frames are shown in rapid succession, they create the illusion of movement.

NEW TERM *frame-based animation*—A form of animation that simulates movement by displaying a sequence of pregenerated, static frame images.

Frame-based animation has no concept of a graphical object distinguishable from the background; everything appearing in a frame is part of that frame as a whole. The result is that each frame image contains all the information necessary for that frame in a static form. This is an important point because it distinguishes frame-based animation from cast-based animation, which you learn about next.

Cast-based Animation

A more powerful animation technique that is often employed by games is *cast-based animation*, which is also known as *sprite animation*. Cast-based animation involves graphical objects that move independently of a background. At this point, you might be a little confused by the use of the term graphical object when referring to parts of an animation.

In this case, a graphical object is something that logically can be thought of as a separate entity from the background of an animation image. For example, in the animation of a space shoot-em-up game, the aliens are separate graphical objects that are logically independent of the starfield background.

NEW TERM *cast-based animation*—A form of animation that simulates movement using graphical objects that move independently of a background.

Each graphical object in a cast-based animation is referred to as a *sprite*, and can have a position that varies over time. In other words, sprites have a velocity associated with them that determines how their position changes over time. Almost every classic video game uses sprites to some degree. For example, every object in the classic Asteroids game is a sprite that moves independently of the background.

NEW TERM *sprite*—A graphical object that can move independently of a background or other objects.

Note

You might be wondering where the term cast-based animation comes from. Sprites can be thought of as cast members moving around on a stage. This analogy of relating computer animation to theatrical performance is very useful. By thinking of sprites as cast members and the background as a stage, you can take the next logical step and think of an animation as a theatrical performance. In fact, this isn't far from the mark, because the goal of theatrical performances is to entertain the audience by telling a story through the interaction of the cast members. Likewise, cast-based animations use the interaction of sprites to entertain the user while often telling a story.

Although the fundamental principle behind sprite animation is the positional movement of a graphical object, you can incorporate frame-based animation into a sprite. Incorporating frame-based animation into a sprite enables you to change the image of the sprite as well as alter its position. This hybrid type of animation is actually what you will implement later in today's lesson in the MIDlet sprite classes.

Television is a good example of frame-based animation. Think of something on television that is created in a manner similar to cast-based animation (other than animated movies and cartoons). Have you ever wondered how weather forecasters magically appear in front of a computer-generated map showing the weather? Newscasters use a technique known as *blue-screening*, which enables them to overlay the weather forecaster on top of the weather map in real-time. It works like this: The person stands in front of a blue backdrop, which serves as a transparent background. The image of the weather forecaster is overlaid onto the weather map; the trick is that the blue background is filtered out when the image is overlaid so that it is effectively transparent. In this way, the weather forecaster acts exactly like a sprite!

Transparency

The weather forecaster example highlights a very important point regarding sprites: *transparency*. Because bitmapped images are rectangular by nature, a problem arises when sprite images aren't rectangular in shape. With sprites that aren't rectangular in shape—the majority of sprites—the pixels surrounding the sprite image are unused. In a graphics system without transparency, these unused pixels are drawn just like any others. The end result is sprites that have visible rectangular borders around them, which completely destroys the effectiveness of having sprites overlaid on a background image.

One solution is to make all of your sprites rectangular. Because this solution isn't very practical, a more realistic solution is transparency, which enables you to define a certain color in an image as unused, or transparent. When pixels of this color are encountered by drawing routines, they are simply skipped, leaving the background visible underneath. Transparent colors in images act exactly like the weathercaster's blue screen in the previous example.

NEW TERM *transparent color*—A color in an image that isn't drawn when the rest of the colors in the image are drawn.

You're probably thinking that implementing transparency involves a lot of low-level bit twiddling and image pixel manipulation. In some programming environments, you would be correct in this assumption, but not in Java. Fortunately, transparency is already supported in Java by way of the GIF and PNG image formats. Using either of these image formats, you simply specify a color of the image that serves as the transparent color. When the image is drawn, pixels matching the transparent color are skipped and left undrawn, leaving the background pixels in plain view.

Z-Order

In many instances, you will want some sprites to appear on top of others. For example, in a war game you might have planes flying over a battlefield dropping bombs on everything in sight. If a plane sprite happens to fly over a tank sprite, you obviously want the plane to appear above the tank and, therefore, hide the tank as it passes over. You handle this problem by assigning each sprite a screen depth, which is also referred to as *Z-order*.

NEW TERM *Z-order*—The relative depth of sprites with respect to the surface of the screen.

The depth of sprites is called Z-order because it works sort of like another dimension—like a Z axis. You can think of sprites moving around on the screen in the XY axis.

Similarly, the Z-axis can be thought of as another axis projected into the screen that determines how the sprites overlap each other. To put it another way, Z-order determines a sprite's depth within the screen. By making use of a Z-axis, you might think that Z-ordered sprites are 3D. The truth is that Z-ordered sprites can't be considered 3D because the Z-axis is a hypothetical axis that is only used to determine how sprite objects hide each other.

To make sure that you have a clear picture of how Z-order works, think for a moment of the good old days of traditional animation. Traditional animators, such as those at Disney, used celluloid sheets to draw animated objects. They drew on celluloid sheets because the sheets could be overlaid on a background image and moved independently. This was known as *cel animation* and should sound vaguely familiar. (Cel animation is an early version of sprite animation.) Each cel sheet corresponds to a unique Z-order value, determined by where in the pile of sheets the sheet is located. If a sprite near the top of the pile happens to be in the same location on the cel sheet as any lower sprites, it conceals them. The location of each sprite in the stack of cel sheets is its Z-order, which determines its visibility precedence. The same thing applies to sprites in cast-based animations, except that the Z-order is determined by the order in which the sprites are drawn, rather than the cel sheet location. This concept of a pile of cel sheets representing all the sprites in a sprite system will be useful later today when you develop the sprite classes.

Collision Detection

No discussion of animation would be complete without covering collision detection. *Collision detection* is the method of determining whether sprites have collided with each other. Although collision detection doesn't directly play a role in creating the illusion of movement, it is tightly linked to sprite animation, and is extremely crucial in games.

NEW TERM *collision detection*—The process of determining whether sprites have collided with each other.

Collision detection determines when sprites physically interact with each other. In an Asteroids game, for example, if the ship sprite collides with an asteroid sprite, the ship is destroyed. Collision detection is the mechanism employed to find out whether the ship collided with the asteroid. This might not sound like a big deal; just compare their positions and see whether they overlap, right? Correct, but consider how many comparisons must take place when many sprites are moving around; each sprite must be compared to every other sprite in the system. It's not hard to see how the processing overhead of collision detection can become difficult to manage.

Not surprisingly, many approaches exist to handling collision detection. The simplest approach is to compare the bounding rectangles of each sprite with the bounding rectangles of all the other sprites. This method is very efficient, but if you have objects that are not rectangular, a certain degree of error occurs when the objects brush by each other. This is because the corners might overlap and indicate a collision when really only the transparent areas are overlapping. The more irregular the shape of the sprites, the more error typically occurs.

An improvement on this technique is to shrink the collision rectangles a little, which reduces the corner error. This method improves things a little, but it has the potential of causing error in the reverse direction by allowing sprites to overlap in some cases without signaling a collision. Shrunken rectangle collision is just as efficient as simple rectangle collision because all you do is compare rectangles for intersection.

The most accurate collision detection technique is to detect collisions based on the sprite image data, which involves actually checking to see whether transparent parts of the sprite or the sprite images themselves are overlapping. In this case, you get a collision only if the actual sprite images are overlapping. Unfortunately, this technique requires far more overhead than rectangle collision detection and is often a major bottleneck in performance. Furthermore, implementing image data for collision detection can get very messy.

Implementing Sprite Animation

You have learned that sprite animation involves the movement of individual graphic objects called sprites. Unlike simple frame animation, sprite animation involves considerably more overhead. More specifically, it is necessary to develop not only a sprite class, but also a sprite management class for keeping up with all the sprites in the system. This is necessary because sprites must be able to interact with each other through a common mechanism. Furthermore, it is useful to extract the background behind the sprites into a class of its own.

In this section, you will learn how to implement sprite animation for MIDlets by creating a suite of sprite classes. The primary sprite classes are `Sprite` and `SpriteVector`. However, you will also learn about a few support classes as you learn the details of these two primary classes. The `Sprite` class models a single sprite and contains all the information and methods necessary to get a single sprite up and running. However, the real power of sprite animation is harnessed by combining the `Sprite` class with the `SpriteVector` class, which is a container class that manages multiple sprites and their interaction with each other.

The `Sprite` Class

Although sprites can be implemented simply as movable graphical objects, the sprite class developed here will also support frame animation. A frame-animated sprite is basically a sprite with multiple frame images that can be displayed in succession. The `Sprite` class you are about to work through supports frame animation in the form of an array of frame images and some methods for setting the current frame image. Using this approach, you end up with a `Sprite` class that supports both fundamental types of animation and is much more suitable for heavy-duty animated MIDlets such as games.

Before jumping into the details of how the `Sprite` class is implemented, think about the different pieces of information that a sprite must keep up with. When you understand the components of a sprite at a conceptual level, it will be much easier to understand the Java code. So, exactly what information should the `Sprite` class maintain? The following list contains the key information that the `Sprite` class needs to include:

- Array of frame images
- Current frame
- XY position
- Width and height
- Velocity
- Z-order

The first item, an array of frame images, is necessary to carry out the frame animations. Even though this sounds as if you are forcing a sprite to have multiple animation frames, a sprite can also use a single image. In this way, the frame animation aspects of the sprite are optional. The current frame keeps up with the current frame of animation. In a typical frame-animated sprite, the current frame is incremented to the next frame when the sprite is updated.

The XY position stores the position of the sprite. You can move the sprite by altering this position. Alternatively, you can set the velocity and let the sprite alter its position internally. The width and height of the sprite can be combined with the XY position to form a boundary rectangle for the sprite. All sprites are bounded by some region, which is usually the size of the MIDlet screen. The sprite boundary is important because it determines the limits of a sprite's movement. Finally, the Z-order represents the depth of the sprite in relation to other sprites. Ultimately, the Z-order of a sprite determines its drawing order (more on that a little later).

Now that you understand the core information required by the `Sprite` class, it's time to get into the specific implementation. Let's begin with the `Sprite` class's member variables, which follow in Listing 17.1.

LISTING 17.1 The `Sprite` Class Member Variables

```
// Sprite action flags
public static final int SA_KILL = 1,
                        SA_RESTOREPOS = 2,
                        SA_ADDSPRITE = 4;
// Bounds actions
public static final int BA_STOP = 0,
                        BA_WRAP = 1,
                        BA_BOUNCE = 2,
                        BA_DIE = 3;
protected Image[]       image;
protected int           frame,
                        frameInc,
                        frameDelay,
                        frameTrigger;
protected int           xPosition, yPosition,
                        width, height;
protected int           zOrder;
protected int           xVelocity, yVelocity;
protected int           xBounds, yBounds, wBounds, hBounds;
protected int           boundsAction;
protected boolean       hidden = false;
```

The member variables include the important sprite information mentioned earlier, along with some other useful information. Most notably, you are probably curious about the static final members at the beginning of the listing. These members are constant identifiers that define actions for the sprite. Two different types of actions are supported by `Sprite`: sprite actions and bounds actions. Sprite actions are general actions that a sprite can perform, such as killing itself or adding another sprite. Bounds actions are actions that a sprite takes in response to reaching a boundary, such as wrapping to the other side or bouncing. Unlike sprite actions, bounds actions are mutually exclusive, meaning that only one can be set at a time.

The `frameInc` member variable is used to provide a means to change the way that the animation frames are updated. In some cases you might want the frames to be displayed in the reverse order. You can easily do this by setting `frameInc` to -1 (its typical value is 1). The `frameDelay` and `frameTrigger` member variables are used to provide a means of varying the speed of the frame animation. You'll see how the speed of animation is controlled when you learn about the `incFrame` method a little later in the lesson.

The last member variable, `hidden`, is a Boolean flag that determines whether the sprite is hidden. When you set this variable to `true`, the sprite is hidden from view. Its default setting is `false`, meaning that the sprite is visible by default.

These member variables are initialized in the two `Sprite` constructors. The first constructor creates a `Sprite` without frame animations, that is, it uses a single image to represent the sprite. The code for this constructor is shown in Listing 17.2.

LISTING 17.2 The Basic `Sprite` Constructor Without Support for Frame Animation

```
public Sprite(Image img, int xPos, int yPos, int xVel, int yVel,
  int z, int wBnds, int hBnds, int ba) {
  image = new Image[1];
  image[0] = img;
  setPosition(xPos, yPos);
  width = img.getWidth();
  height = img.getHeight();
  setVelocity(xVel, yVel);
  frame = 0;
  frameInc = 0;
  frameDelay = frameTrigger = 0;
  zOrder = z;
  xBounds = 0;
  yBounds = 0;
  wBounds = wBnds;
  hBounds = hBnds;
  boundsAction = ba;
}
```

This constructor takes an image, position, velocity, Z-order, boundary, and boundary action as parameters. The second constructor takes an array of images and some additional information about the frame animations. The code for the second constructor is shown in Listing 17.3.

LISTING 17.3 A `Sprite` Constructor That Supports Frame Animation

```
public Sprite(Image[] img, int f, int fi, int fd, int xPos,
  int yPos, int xVel, int yVel, int z, int wBnds, int hBnds,
  int ba) {
  image = img;
  setPosition(xPos, yPos);
  width = img[f].getWidth();
  height = img[f].getHeight();
  setVelocity(xVel, yVel);
  frame = f;
  frameInc = fi;
  frameDelay = frameTrigger = fd;
  zOrder = z;
  xBounds = 0;
  yBounds = 0;
```

LISTING 17.3 continued

```
    wBounds = wBnds;
    hBounds = hBnds;
    boundsAction = ba;
  }
```

The additional information required of this constructor includes the current frame (`f`), frame increment (`fi`), and frame delay (`fd`).

The `Sprite` class contains a number of access methods, which are simply interfaces to get and set certain member variables. These methods consist of one or two lines of code and are fairly self-explanatory. Check out the code for the `getXVelocity()`, `getYVelocity()`, and `setVelocity()` access methods to see. These are included in Listing 17.4.

LISTING 17.4 The `Sprite` Access Methods

```
public int getXVelocity() {
  return xVelocity;
}

public int getYVelocity() {
  return yVelocity;
}

public void setVelocity(int xVel, int yVel) {
  xVelocity = xVel;
  yVelocity = yVel;
}
```

More access methods exist for getting and setting other member variables in the `Sprite` class, but they are just as straightforward as the velocity access methods. Rather than spending time on those, let's move on to some more interesting methods!

The `incFrame()` method is the first `Sprite` method with any real substance, as is shown in Listing 17.5.

LISTING 17.5 The `incFrame()` Method Increments the Animation Frame of the Sprite

```
protected void incFrame() {
  if ((frameDelay > 0) && (--frameTrigger <= 0)) {
    // Reset the frame trigger
    frameTrigger = frameDelay;
```

LISTING 17.5 continued

```
      // Increment the frame
      frame += frameInc;
      if (frame >= image.length)
        frame = 0;
      else if (frame < 0)
        frame = image.length - 1;
    }
  }
```

The `incFrame()` method is used to increment the current animation frame. It first checks the `frameDelay` and `frameTrigger` member variables to see whether the frame should actually be incremented. This check is what enables you to vary the frame animation speed for a sprite, which is accomplished by changing the value of `frameDelay`. Larger values for `frameDelay` result in a slower animation speed. The current frame is incremented by adding `frameInc` to `frame`. `frame` is then checked to make sure that its value is within the bounds of the image array because it is used later to index into the array when the frame image is drawn.

The method that does most of the work in the `Sprite` class is the `update()` method, whose code follows in Listing 17.6.

LISTING 17.6 The `update()` Method Updates the Position of the Sprite and Increments Its Animation Frame

```
public int update() {
  int action = 0;

  // Increment the frame
  incFrame();

  // Update the position
  int xPos = xPosition;
  int yPos = yPosition;
  xPos += xVelocity;
  yPos += yVelocity;

  // Check the bounds
  // Wrap?
  if (boundsAction == Sprite.BA_WRAP) {
    if ((xPos + width) < xBounds)
      xPos = xBounds + wBounds;
    else if (xPos > (xBounds + wBounds))
      xPos = xBounds - width;
    if ((yPos + height) < yBounds)
```

LISTING 17.6 continued

```
          yPos = yBounds + hBounds;
        else if (yPos > (yBounds + hBounds))
          yPos = yBounds - height;
      }
      // Bounce?
      else if (boundsAction == Sprite.BA_BOUNCE) {
        boolean bounce = false;
        int xVel = xVelocity;
        int yVel = yVelocity;
        if (xPos < xBounds) {
          bounce = true;
          xPos = xBounds;
          xVel = -xVel;
        }
        else if ((xPos + width) >
          (xBounds + wBounds)) {
          bounce = true;
          xPos = xBounds + wBounds - width;
          xVel = -xVel;
        }
        if (yPos < yBounds) {
          bounce = true;
          yPos = yBounds;
          yVel = -yVel;
        }
        else if ((yPos + height) >
          (yBounds + hBounds)) {
          bounce = true;
          yPos = yBounds + hBounds - height;
          yVel = -yVel;
        }
        if (bounce)
          setVelocity(xVel, yVel);
      }
      // Die?
      else if (boundsAction == Sprite.BA_DIE) {
        if ((xPos + width) < xBounds ||
          xPos > wBounds ||
          (yPos + height) < yBounds ||
          yPos > hBounds) {
          action |= Sprite.SA_KILL;
          return action;
        }
      }
      // Stop (default)
      else {
        if (xPos < xBounds ||
          xPos > (xBounds + wBounds - width)) {
          xPos = Math.max(xBounds, Math.min(xPos,
```

Listing 17.6 continued

```
        xBounds + wBounds - width));
      setVelocity(0, 0);
    }
    if (yPos  < yBounds ||
      yPos > (yBounds + hBounds - height)) {
      yPos = Math.max(yBounds, Math.min(yPos,
        yBounds + hBounds - height));
      setVelocity(0, 0);
    }
  }
  setPosition(xPos, yPos);

  return action;
}
```

The `update()` method handles the task of updating the animation frame and position of the sprite. The method begins by creating an empty set of action flags that are stored as individual bit flags of an integer. The animation frame is then updated with a call to `incFrame()`. The position of the sprite is updated by translating the position based on the velocity. You can think of the position as sliding a distance determined by the velocity.

The rest of the code in the `update()` method is devoted to handling the various bounds actions. The first bounds action flag, `BA_WRAP`, causes the sprite to wrap around to the other side of the bounds rectangle. This flag is useful in an Asteroids-type game, in which the asteroids float off one side of the screen and back from the other. The `BA_BOUNCE` flag causes the sprite to bounce if it encounters a boundary. This flag is useful in a Breakout or Pong-type game, in which a ball bounces off the edges of the screen. The `BA_DIE` flag causes the sprite to die if it encounters a boundary. This flag is useful for sprites such as bullets, which you often want to be destroyed when they travel beyond the edges of the screen. Finally, the default flag, `BA_STOP`, causes the sprite to stop when it encounters a boundary.

Notice that the `update()` method finishes by returning an integer containing the sprite action flags. Derived sprite classes can return different sprite action values to trigger different actions. Judging by its size, it's not hard to determine that the `update()` method is itself the bulk of the code in the `Sprite` class. This is logical though, because the `update()` method is where all the action takes place; `update()` handles all the details of updating the animation frame and position of the sprite, along with carrying out different bounds actions.

Another important method in the Sprite class is draw(), whose source code is in Listing 17.7.

LISTING 17.7 The draw() Method Draws the Sprite Image

```
public void draw(Graphics g) {
  // Draw the current frame
  if (!hidden)
    g.drawImage(image[frame], xPosition, yPosition,
      Graphics.TOP | Graphics.LEFT);
}
```

After wading through the update() method, the draw() method probably looks like a piece of cake! It uses the drawImage() method to draw the current sprite frame image to the Graphics object that is passed in. Notice that the drawImage() method requires the image, XY position, and anchor information to carry this out.

Another interesting method in the Sprite class is the addSprite() method, which is suspiciously simple, as the following code reveals:

```
protected Sprite addSprite(int action) {
  return null;
}
```

To understand the purpose of the addSprite() method, you need to know that a sprite list that contains all the sprites exists and is maintained by the SpriteVector class, which you learn about a little later today. The addSprite() method is needed because a sprite occasionally must create and add another sprite to the sprite list. A big problem exists because an individual sprite doesn't know anything about the sprite list. To get around this problem, you use sprite actions. Sprite actions work like this: A sprite notifies the sprite list that it wants to add a sprite by setting the SA_ADDSPRITE action flag in the set of action flags returned by the update() method. The sprite list, in turn, calls the addSprite() method for the sprite and adds the new sprite to the list. This sounds like a convoluted way to handle sprite creation, but it actually works quite well and fits in with the object-oriented design of the sprite classes. The remaining question, then, is why does this implementation of addSprite() return null? The answer is that it is up to derived sprites to provide a specific implementation of the addSprite() method. Knowing this, you could make addSprite() abstract, but then you would be forced to derive a new sprite class any time you want to create a sprite.

The last method of significance in Sprite is testCollision(), which is used to check for collisions between sprites (see Listing 17.8).

LISTING 17.8 The `testCollision()` Method Tests for a Collision with Another Sprite

```
protected boolean testCollision(Sprite test) {
  // Check for collision with another sprite
  if (test != this)
    return intersects(test.getXPosition(), test.getYPosition(),
      test.getWidth(), test.getHeight());
  return false;
}
```

The sprite to test for collision is passed in as the `test` parameter. The test simply involves checking to see whether the boundary rectangles of the sprites intersect. If so, `testCollision()` returns `true`. It's worth noting that the `testCollision()` method isn't very useful within the context of a single sprite, but it is very handy when you put it together with the `SpriteVector` class, which you learn about next.

Before moving on to the `SpriteVector` class, there is one last bit of unfinished business—the `intersects()` method, which is called by the `testCollision()` method. The `intersects()` method checks to see whether a specified sprite boundary intersects the current sprite's boundary. The code for the `intersects()` method is in Listing 17.9.

LISTING 17.9 The `intersects()` Method Checks to See Whether the Boundaries of Two Sprites Intersect

```
protected boolean intersects(int xTest, int yTest, int wTest, int hTest) {
  // Check for intersection with another sprite
  return
    ((xTest > xPosition && xTest < (xPosition + width)) ||
    (xPosition > xTest && xPosition < (xTest + wTest))) &&
    ((yTest > yPosition && yTest < (yPosition + height)) ||
    (yPosition > yTest && yPosition < (yTest + hTest)));
}
```

Although this method looks a bit messy, the code is actually very simple if you study it for a moment. The return value is calculated by testing the upper-left corner of each sprite to see if it is positioned within the bounds of the other sprite.

The `SpriteVector` Class

At this point, you have a `Sprite` class with some impressive features, but you don't really have any way to manage it. Of course, you could go ahead and create a MIDlet with some `Sprite` objects, but how would they be able to interact with each other? The answer to this question is the `SpriteVector` class, which handles all the details of maintaining a list of sprites and the interactions between them.

The SpriteVector class is derived from the Vector class, which is a standard MIDP collection class provided in the java.util package. The Vector class models a growable array of objects. In this case, the SpriteVector class is used as a container for a growable array of Sprite objects.

Getting into the implementation of the SpriteVector class, there is only one member variable, background, which is an object of type Background:

```
protected Background background;
```

The Background object represents the background upon which the sprites appear. You learn about the Background class a little later in the lesson. The background member variable is initialized in the constructor for SpriteVector, as shown in the following code:

```
public SpriteVector(Background back) {
  super(20, 5);
  background = back;
}
```

The constructor for SpriteVector takes a Background object as its only parameter. Notice that the SpriteVector() constructor calls the Vector parent class constructor (super()) and sets the default storage capacity to 20 and the amount to increment the storage capacity to 10 if the vector needs to grow. It's reasonable to expect that 20 is on the high end of the number of sprites used in a given MIDlet, because of the limited resources of the MIDP environment.

The SpriteVector class contains the following two access methods for getting and setting the background member variable (see Listing 17.10).

LISTING 17.10 The getBackground() and setBackground() Methods Provide Access to the Background of the Sprite Vector

```
public Background getBackground() {
  return background;
}

public void setBackground(Background back) {
  background = back;
}
```

These methods are useful in games where the background changes based on the level of the game. To change the background, you call setBackground() and pass in the new Background object.

As in the Sprite class, the update() method is the key method in the SpriteVector class because it handles updating all the sprites. The code for update() is in Listing 17.11.

LISTING 17.11 The SpriteVector update() Method Updates All of the Sprites in the Sprite Vector

```
public void update() {
  // Iterate through sprites, updating each
  Sprite    s, sHit;
  int       xLastPos, yLastPos;
  for (int i = 0; i < size(); ) {
    // Update the sprite
    s = (Sprite)elementAt(i);
    xLastPos = s.getXPosition();
    yLastPos = s.getYPosition();
    int action = s.update();

    // Check for the SA_ADDSPRITE action
    if ((action & Sprite.SA_ADDSPRITE) == Sprite.SA_ADDSPRITE) {
      // Add the sprite
      Sprite sToAdd = s.addSprite(action);
      if (sToAdd != null) {
        int iAdd = add(sToAdd);
        if (iAdd >= 0 && iAdd <= i)
          i++;
      }
    }

    // Check for the SA_RESTOREPOS action
    if ((action & Sprite.SA_RESTOREPOS) == Sprite.SA_RESTOREPOS)
      s.setPosition(xLastPos, yLastPos);

    // Check for the SA_KILL action
    if ((action & Sprite.SA_KILL) == Sprite.SA_KILL) {
      removeElementAt(i);
      continue;
    }

    // Test for collision
    int iHit = testCollision(s);
    if (iHit >= 0)
      if (collision(i, iHit))
        s.setPosition(xLastPos, yLastPos);
    i++;
  }
}
```

The `update()` method iterates through the sprites, calling the `Sprite` class `update()` method on each individual sprite. It then checks for the various sprite action flags returned by each call to `update()`. If the `SA_ADDSPRITE` flag is set, the `addSprite()` method is called on the sprite and the returned sprite is added to the list. If the `SA_RESTOREPOS` flag is set, the sprite position is set to the position of the sprite prior to being updated. If the `SA_KILL` flag is set, the sprite is removed from the sprite list. Finally, the `testCollision()` method is called to see whether a collision has occurred between sprites. You get the whole scoop on the `testCollision()` method in just a moment. If a collision has occurred, the old position of the collided sprite is restored and the `collision()` method is called to respond to the collision.

The `collision()` method is used to handle collisions between two sprites and take any necessary actions, as shown in Listing 17.12.

LISTING 17.12 The `collision()` Method Handles a Collision Between Two Sprites

```
protected boolean collision(int i, int iHit) {
  // Swap velocities (bounce)
  Sprite s = (Sprite)elementAt(i);
  Sprite sHit = (Sprite)elementAt(iHit);
  int xSwap = s.getXVelocity();
  int ySwap = s.getYVelocity();
  s.setVelocity(sHit.getXVelocity(), sHit.getYVelocity());
  sHit.setVelocity(xSwap, ySwap);
 return true;
}
```

The `collision()` method is responsible for handling any actions that result from a collision between sprites. The action in this case is to swap the velocities of the collided `Sprite` objects, which results in a bouncing effect. This method is where you provide specific collision actions in derived sprites. In a space game, you might want alien sprites to explode upon collision with meteor sprites.

The `testCollision()` method is used to test for collisions between a sprite and the rest of the sprites in the sprite list (see Listing 17.13).

LISTING 17.13 The `testCollision()` Method Tests for Collisions Among Sprites in the Sprite Vector

```
protected int testCollision(Sprite test) {
  // Check for collision with other sprites
  Sprite  s;
  for (int i = 0; i < size(); i++)
```

LISTING 17.13 continued

```
    {
      s = (Sprite)elementAt(i);
      if (s == test)  // don't check itself
        continue;
      if (test.testCollision(s))
        return i;
    }
    return -1;
  }
```

The sprite to be tested for collision is passed in the `test` parameter. The sprites are then iterated through, and the `testCollision()` method in the `Sprite` class is called for each sprite. Notice that the `testCollision()` method isn't called on the test sprite if the iteration refers to the same sprite. To understand the significance of this code, consider the effect of passing `testCollision()` the same sprite on which the method is being called; you would check to see whether a sprite was colliding with itself, which would always return `true`. If a collision is detected, the `Sprite` object that has been hit is returned from `testCollision()`.

The `SpriteVector` `draw()` method handles drawing the background, as well as drawing all the sprites (see Listing 17.14).

LISTING 17.14 The `SpriteVector` `draw()` Method Draws All of the Sprites in the Sprite Vector

```
public void draw(Graphics g) {
  // Draw the background
  background.draw(g);

  // Iterate through sprites, drawing each
  for (int i = 0; i < size(); i++)
    ((Sprite)elementAt(i)).draw(g);
}
```

The background is drawn with a quick call to the `draw()` method of the `Background` object. The sprites are then drawn by iterating through the sprite list and calling the `draw()` method on each of the sprites.

The `add()` method is without a doubt the trickiest method in the `SpriteVector` class, as Listing 17.15 reveals.

LISTING 17.15 The `SpriteVector` `add()` Method Adds a Sprite to the Sprite Vector

```
public int add(Sprite s) {
  // Use a binary search to find the right location to insert the
  // new sprite (based on Z-order)
  int   l = 0, r = size(), i = 0;
  int   z = s.getZOrder(),
        zTest = z + 1;
  while (r > l) {
    i = (l + r) / 2;
    zTest = ((Sprite)elementAt(i)).getZOrder();
    if (z < zTest)
      r = i;
    else
      l = i + 1;
    if (z == zTest)
      break;
  }
  if (z >= zTest)
    i++;

  insertElementAt(s, i);
  return i;
}
```

The `add()` method handles adding new sprites to the sprite list, which doesn't sound like a very difficult task. The catch is that the sprite list must always be sorted according to Z-order. Why? Z-order is the depth at which sprites appear onscreen. The illusion of depth is established by the order in which the sprites are drawn. This works because sprites drawn later are drawn on top of sprites drawn earlier, and therefore appear to be at a higher depth. Therefore, sorting the sprite list by ascending Z-order and then drawing them in that order is an effective way to provide the illusion of depth. The `add()` method uses a binary search to quickly find the correct spot to add new sprites so that the sprite list remains sorted by Z-order.

That wraps up the `SpriteVector` class. You now have not only a powerful `Sprite` class, but also a `SpriteVector` class for managing and providing interactivity between sprites. You are one step away from being able to use these classes in your first animated MIDlet.

The Background Classes

When developing the `SpriteVector` class you came into contact with the `Background` class, which is used to represent a background for sprites. The next few sections reveal the inner workings of this class, along with a few other background classes that you will find useful in future animated MIDlets.

The `Background` Class

As you now know, the `Background` class provides the overhead of managing a background for the sprites to appear on top of. Listing 17.16 shows the source code for the `Background` class.

LISTING 17.16 The `Background` Class Represents a Generic Background for a Sprite Vector

```
public class Background {
  protected int width, height;

  public Background(int w, int h) {
    width = w;
    height = h;
  }

  public int getWidth() {
    return width;
  }

  public int getHeight() {
    return height;
  }

  public void draw(Graphics g) {
    // Save the old color
    int color = g.getColor();

    // Fill with white
    g.setColor(255, 255, 255);
    g.fillRect(0, 0, width, height);

    // Restore the old color
    g.setColor(color);
  }
}
```

The `Background` class basically provides a no-frills white background for sprites. The two member variables maintained by `Background` are used to keep up with the width and height of the background. The constructor for `Background` takes the width and height of the background as its only parameters. The `getWidth()` and `getHeight()` methods are access methods that simply return the size of the background. The `draw()` method fills the background with white, which is why I referred to the `Background` class as providing a no-frills background for sprites.

You're probably thinking that this `Background` object isn't too exciting. Couldn't you just stick this drawing code directly into the `SpriteVector` class's `draw()` method? Yes, you could, but then you would miss out on the benefits provided by the derived background classes, `ColorBackground` and `ImageBackground`, which are explained next. The background classes are a good example of how object-oriented design makes Java code much cleaner and easier to extend.

The `ColorBackground` Class

The `ColorBackground` class is derived from `Background`, and provides a background that can be filled with any color. The code for the `ColorBackground` class is shown in Listing 17.17.

LISTING 17.17 The `ColorBackground` Class Represents a Color Background

```
public class ColorBackground extends Background {
  protected int red, green, blue;

  public ColorBackground(int w, int h, int r, int g, int b) {
    super(w, h);
    setColor(r, g, b);
  }

  public int getRed() {
    return red;
  }

  public int getGreen() {
    return green;
  }

  public int getBlue() {
    return blue;
  }

  public void setColor(int r, int g, int b) {
    red = r;
    green = g;
    blue = b;
  }

  public void draw(Graphics g) {
    // Save the old color
    int color = g.getColor();
```

LISTING 17.17 continued

```
    // Fill with the color
    g.setColor(red, green, blue);
    g.fillRect(0, 0, width, height);

    // Restore the old color
    g.setColor(color);
  }
}
```

The `ColorBackground` class adds a few member variables to represent the color of the background. The constructor for `ColorBackground` takes the three components of the background color as parameters. Several access methods exist for getting and setting the color of the background. The `draw()` method for the `ColorBackground` class is very similar to the `draw()` method in `Background` except that the `red`, `green`, and `blue` color member variables are used as the background fill color.

The `ImageBackground` Class

A more interesting `Background`-derived class is `ImageBackground`, which uses an image as the background. The source code for the `ImageBackground` class is shown in Listing 17.18.

LISTING 17.18 The `ImageBackground` Class Represents an Image Background

```
public class ImageBackground extends Background {
  protected Image image;

  public ImageBackground(Image img) {
    super(img.getWidth(), img.getHeight());
    image = img;
  }

  public Image getImage() {
    return image;
  }

  public void setImage(Image img) {
    image = img;
  }

  public void draw(Graphics g) {
    // Draw background image
    g.drawImage(image, 0, 0, Graphics.TOP | Graphics.LEFT);
  }
}
```

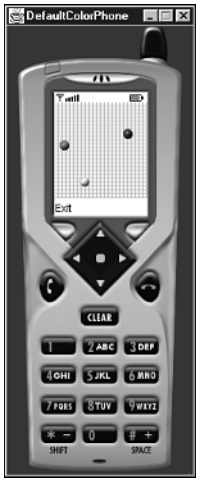
The `ImageBackground` class adds a single member variable, `image`, which is an `Image` object. This member variable holds the image to be used as the background. Not surprisingly, the constructor for `ImageBackground` takes an `Image` object as its only parameter. Two access methods exist for getting and setting the `image` member variable. The `draw()` method for the `ImageBackground` class draws the background image using the `drawImage()` method of the passed `Graphics` object.

The Atoms Sample MIDlet

This has been one of the most code-intensive lessons you've encountered in the book, but I promise all the work has a due payoff. It's time to take all the hard work that you put into the sprite classes and see what it amounts to. Figure 17.1 is a screen shot of the Atoms MIDlet, which shows the sprite classes you've toiled over for so long.

Note As with all of the examples throughout the book, the complete source code, resources, and executable classes for the Atoms MIDlet are on the accompanying CD-ROM.

FIGURE 17.1
The Atoms example MIDlet demonstrates how to put the sprite classes to work.



The Animation Canvas

Because sprite animation requires you to be able to draw directly on the screen of a MIDlet, the Atoms MIDlet requires a `Canvas`-derived class that in turn manages its own `SpriteVector` class. This class is called `AtomsCanvas`, and contains virtually all of the animation code for the MIDlet. The `SpriteVector` object for the animation, `sv`, is created as a member variable of the `AtomsCanvas` class. The member variables for this class are included in Listing 17.19.

Listing 17.19 The `AtomsCanvas` Member Variables

```
  private Image        offImage, back;
  private Image[]      atom;
  private Graphics     offGrfx;
  private SpriteVector sv;
  protected Timer      animTimer;
  private int          animPeriod = 83; // 12 fps
  private Random rand = new Random(System.currentTimeMillis());
```

The `Image` member variables in the `AtomsCanvas` class represent the offscreen buffer, the background image, and the atom images. The `Graphics` member variable, `offGrfx`, holds the graphics context for the offscreen buffer image, which you learn about in a moment. The `SpriteVector` member variable, `sv`, holds the sprite vector for the MIDlet. The `animTimer` and `animPeriod` variables are used to control the timing of the animation. Finally, the `Random` member variable, `rand`, is used to generate random numbers throughout the MIDlet.

Notice that the `animPeriod` member variable is set to 83. The `animPeriod` member variable specifies the amount of time (in milliseconds) that elapses between each frame of animation. You can determine the frame rate by inverting the value of delay, which results in a frame rate of about 12 frames per second (fps) in this case. This frame rate is pretty much the minimum rate required for fluid animation, such as sprite animation. You'll see how delay is used to establish the frame rate later in the lesson when you get into the details of the animation task.

The `AtomsCanvas()` constructor is the heart of the Atoms MIDlet because it loads all the images, creates the sprite vector and the sprites, and establishes the animation timer. The following code reveals how the constructor accomplishes these tasks (see Listing 17.20).

Listing 17.20 The `AtomsCanvas` Constructor

```
public AtomsCanvas() {
  // Load the background and atom images
  try {
    back = Image.createImage("/Back.png");
    atom = new Image[6];
    atom[0] = Image.createImage("/Red.png");
    atom[1] = Image.createImage("/Green.png");
```

LISTING 17.20 continued

```
      atom[2] = Image.createImage("/Blue.png");
      atom[3] = Image.createImage("/Yellow.png");
      atom[4] = Image.createImage("/Purple.png");
      atom[5] = Image.createImage("/Orange.png");
    }
    catch (IOException e) {
      System.err.println("Failed loading images!");
    }

    // Create and add the sprites
    sv = new SpriteVector(new ImageBackground(back));
    for (int i = 0; i < 3; i++)
      sv.add(createAtom(
        Math.abs(rand.nextInt() % (getWidth() - atom[0].getWidth())),
        Math.abs(rand.nextInt() % (getHeight() - atom[0].getHeight())),
        Math.abs(rand.nextInt() % 6)));

    // Create the animation timer and schedule the task
    animTimer = new Timer();
    animTimer.schedule(new AnimationTask(), 0, animPeriod);
  }
```

After loading the animation images, the `AtomsCanvas()` constructor creates the `SpriteVector` object. Three different atom `Sprite` objects are then created by calling the `createAtom()` method within a `for` loop. You learn about the `createAtom()` method a little later today. These atom sprites are added to the sprite vector as they are created.

After creating and adding the sprites, a `Timer` object is created that represents the animation timer. The `Timer` class is perfect for animation because it allows you to establish a periodic timer. In order to implement such a timer, you must derive a class from the `TimerTask` class and implement the `run()` method. This is the method that is called each time a timer event is generated. So, in the case of Atoms, you want to update the animation in the timer task in response to timer events. To establish the timer, you must call the `schedule()` method and pass in a `TimerTask` object, along with the initial delay and the period between timer events. For animation purposes, the initial delay is 0 and the period is the delay between animation frames, which happens to be stored in `animPeriod`.

After the timer is set up, a timer event will be generated every 83 milliseconds, resulting in the `run()` method being called on the `AnimationTask` object. The code for the `AnimationTask` class is shown in Listing 17.21, which reveals the inner workings of this all-important `run()` method.

LISTING 17.21 The `AnimationTask` Class Is Responsible for Updating the Sprite Vector

```
class AnimationTask extends TimerTask {
  public AnimationTask() {
  }

  public void run() {
    // Update everything
    sv.update();
    repaint();
  }
}
```

As you can see, the `AnimationTask` class is extremely simple, as is the `run()` method. The `run()` method basically updates the sprite vector and forces the MIDlet to repaint itself. By forcing a repaint, you are causing the MIDlet to redraw the sprites in their newly updated states. Incidentally, the `run()` method has direct access to the `sv` member variable because the `AnimationTask` class is an inner class of the `AtomsCanvas` class.

The `paint()` method in `AtomsCanvas` is where the sprites are actually drawn to the canvas (see Listing 17.22).

LISTING 17.22 The `AtomsCanvas` `paint()` Method Is Used to Paint the Canvas

```
public void paint(Graphics g) {
  // Create the offscreen graphics context
  if (offGrfx == null) {
    offImage = Image.createImage(getWidth(), getHeight());
    offGrfx = offImage.getGraphics();
  }

  // Draw the sprites
  sv.draw(offGrfx);

  // Draw the offscreen image onto the screen
  g.drawImage(offImage, 0, 0, Graphics.TOP | Graphics.LEFT);
}
```

The `paint()` method uses a technique known as double buffering to eliminate flicker in the sprite animation. *Flicker* is a negative visual effect associated with animation that occurs when a portion of the screen is erased and redrawn rapidly; the screen appears to flicker like a candle. *Double buffering* involves erasing and drawing to an offscreen image, and then drawing the complete results to the screen at once, thereby eliminating flicker. By using double buffering, you eliminate flicker and allow for speedier animations. The `AtomsCanvas` class supports double buffering via the `offImage` member variable, which contains the offscreen buffer image used for drawing the next animation frame. Also, the `offGrfx` member variable contains the graphics context associated with the `offscreen` buffer image.

NEW TERM *flicker*—A negative visual effect associated with animation that occurs when a portion of the screen is erased and redrawn rapidly.

NEW TERM *double buffering*—A technique of rendering animation frames that involves erasing and drawing to an offscreen image, and then drawing the complete results to the screen at once, thereby eliminating flicker.

In the `paint()` method, the offscreen buffer is first created as an `Image` object whose dimensions match those of the canvas. It is important that the offscreen buffer be exactly the same size as the canvas. The graphics context associated with the buffer is then retrieved using the `getGraphics()` method of the `Image` class. After the offscreen buffer is initialized, all you really have to do is tell the `SpriteVector` object to draw itself to the buffer. Remember that the `SpriteVector` object takes care of drawing the background and all the sprites. This is accomplished with a simple call to the sprite vector's `draw()` method. The offscreen buffer is then drawn to the MIDlet window using the `drawImage()` method.

The last method in the `AtomsCanvas` class is `createAtom()`, which handles creating a single atom sprite (see Listing 17.23).

LISTING 17.23 The `createAtom()` Method Creates an Atom Sprite

```
private Sprite createAtom(int xPos, int yPos, int i) {
  return new Sprite(atom[i], xPos, yPos, rand.nextInt() % 5,
    rand.nextInt() % 5, 0, getWidth(), getHeight(), Sprite.BA_BOUNCE);
}
```

The `createAtom()` method takes an XY position as its first two parameters, which determine the sprite's initial position. The third parameter, `i`, is an integer that specifies which atom image to use. `CreateAtom()` then calculates a random velocity for the sprite using the `rand` member variable. Each velocity component for the sprite (X and Y) is given a random value in the range -4–4. The sprite is given a Z-order value of 0. Finally, the sprite is assigned the `BA_BOUNCE` bounds action, which means that it will bounce when it encounters the edge of the MIDlet window.

Putting It All Together

Now that you've seen how much animation code went into the `AtomsCanvas` class, you're probably guessing that there isn't much left for the `Atoms` MIDlet class. You're in fact quite correct with this guess. By placing all of the animation code in the `AtomsCanvas` class, the `Atoms` class is reduced to looking like virtually any other MIDlet. The code for the `Atoms` class is given in Listing 17.24. It is quite bland other than the fact that it creates an `AtomsCanvas` object as its canvas.

LISTING 17.24 The Complete Source Code for the Atoms MIDlet

```
public class Atoms extends MIDlet implements CommandListener {
  private Command exitCommand;
  private Display display;
  private AtomsCanvas animScreen;

  public Atoms() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit command
    exitCommand = new Command("Exit", Command.EXIT, 2);

    // Create the main animation screen form
    animScreen = new AtomsCanvas();

    // Set the Exit command
    animScreen.addCommand(exitCommand);
    animScreen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the animation screen
    display.setCurrent(animScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
    // Kill the animation timer
    animScreen.animTimer.cancel();
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
```

Listing 17.24 continued

```
      destroyApp(false);
      notifyDestroyed();
    }
  }
}
```

You'll notice that an `AtomsCanvas` object is created and set as the MIDlet's screen. From there the animation is carried out automatically using the `AtomsCanvas` class. However, one small bit of code is worth pointing out. In the `destroyApp()` method a call to the `cancel()` method of the `Timer` object controls the animation in the canvas. This method results in the timer being killed, which stops the animation. Although you might think that the timer should automatically stop when the MIDlet is exited, in practice this doesn't always appear to be the case. So, to be safe it is a good idea to explicitly kill the timer as the MIDlet exits. Otherwise, the timer may keep running and eat up system resources.

That's all it takes to get the sprite classes working together in a real MIDlet. Even though there is a decent amount of MIDlet-specific animation code in the `AtomsCanvas` class within the MIDlet, a great deal of functionality is being leveraged from the sprite classes. You can use this MIDlet as a template for other MIDlets you create that use the sprite classes. You now have all the functionality required to manage both cast- and frame-based animation, as well as provide support for interactivity among sprites via collision detection and sprite actions.

Summary

In today's lesson, you learned all about animation, including the two major types of animation: frame-based and cast-based. Adding to this theory, you learned that sprite animation is a particularly useful type of animation. You saw firsthand how to develop a powerful duo of sprite classes for implementing sprite animation, including a few support classes to make the sprite classes more flexible. You then put the sprite classes to work in an example MIDlet that involved relatively little additional overhead.

Although it covered a lot of material, today's lesson laid the groundwork for the next lesson, which uses the sprite classes to create an action game in the same genre as the classic Frogger game. In that lesson, you really see the benefits of having a reusable sprite engine because it frees you to focus on the details of the game itself.

Q&A

Q What exactly is Z-order, and do I really need it?

A Z-order is the depth of a sprite relative to other sprites; sprites with higher Z-order values appear to be on top of sprites with lower Z-order values. You only need Z-order when two or more sprites overlap each other, which is in most games.

Q Why bother with the different types of collision detection?

A The different types of collision detection (rectangle, shrunken rectangle, and image data) provide different trade-offs in regard to performance and accuracy. Rectangle and shrunken rectangle collision detection provide a very high-performance solution, but with moderate to poor accuracy. Image data collision detection is perfect when it comes to accuracy, but it can bring an animation to its knees in the performance department.

Q Why do I need the `SpriteVector` class? Isn't the `Sprite` class enough?

A The `Sprite` class is nice, but it only represents a single sprite. To enable multiple sprites to interact with each other, you must have a second entity that acts as a storage unit for the sprites. The `SpriteVector` class solves this problem by doubling as a container for sprites as well as a communication medium between sprites.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What are the two main types of animation?
2. What is a transparent color?
3. What is flicker?
4. What is double buffering?

Exercises

1. Modify the Atoms MIDlet so that the sprites wrap around the edges of the screen rather than bouncing.
2. Substitute frame animated sprites for the single frame atom sprites. Hint: This primarily involves creating one or more images for each atom sprite that somehow show the atom changing, maybe pulsating? You then use the second `Sprite()` constructor to create the sprites, specifying appropriate frame-related information.

DAY 18

An Ode to Pong: Creating MIDlet Games

In Day 17, “Creating Animated MIDlets,” you created a set of sprite classes that serve as a powerful framework for creating sprite-animated MIDlets. Although sprite animation is pretty neat by itself, it is much more interesting when you use it as the basis for creating games. This lesson picks up where Day 17 ended by showing you how to use the sprite classes to create a fully functional action MIDlet game. The game won’t have you contemplating getting rid of your Xbox or Playstation 2, but it does demonstrate how to create a pretty neat little MIDlet game.

The game you develop in this lesson is called Traveling Gecko, and it is a vague remake of the classic Frogger game. In Traveling Gecko, you must safely guide a little red gecko across the desert past several predators, including tarantulas, rattlesnakes, and Gila monsters. The code for the game is built around the sprite class that you developed in Day 17. You will find the Traveling Gecko game to be a great starting point for developing MIDlet games of your own. The following are the major topics covered in this lesson:

- Brainstorming and designing the Traveling Gecko game
- Creating sprite classes for Traveling Gecko
- Developing the canvas and MIDlet classes for Traveling Gecko
- Testing the Traveling Gecko game

Designing Traveling Gecko

It is easy to get into trouble by writing code without a solid direction, so it is very important to think through a game design as thoroughly as possible before writing any code. Before writing any line of code, spend some time thinking about the design of a game in both general and specific terms. With that in mind, let's break the Traveling Gecko example game into its logical components.

The Traveling Gecko game is modeled on the classic Frogger arcade game. In the original Frogger game, you guide a frog through traffic and then across a river, using floating logs to get across. Traveling Gecko takes a similar approach in that the goal is to maneuver an animal from one place to another while dodging dangers along the way. However, the setting for Traveling Gecko is the Desert Southwest, and your character is a gecko on the move. He wants only to get across a particularly small stretch of desert, but he must contend with a variety of predators to do so.

Sprites

Given the game description thus far, you probably already have in mind some sprites that the game will need. We will go ahead and break the game down into sprites, because that's where most of the substance of the game is located. Obviously, the most important sprite is the gecko sprite itself, which needs to be able to move based on user input. The gecko sprite is the heart of the game and must be designed with care. Because the gecko is capable of being killed by the predators, you'll also need an animation of the gecko dying—a geckocide sprite, if you will. The geckocide sprite shows an animation of the gecko dying so that it doesn't just disappear when it dies. This is a subtle aspect of the game that goes a long way toward making the game more interesting to the user.

It is fairly obvious that you will also need sprite objects for the predators. Although each one has basically the same functionality, think of them as unique sprite objects, because you might decide to add specialized behavior to each of them later. You should have some special logic for handling a collision between the predators and the gecko because this contact results in the gecko's death.

Before we finish itemizing the sprites, think about the specifics surrounding the gecko's path across the desert. Taking an approach similar to the game Frogger, the gecko must travel from the bottom of the screen to safety at the top. However, it seems too easy simply to have him go from the bottom of the screen to the top with no obstacles other than the predators. Frogger has specific locations at the top of the screen where the frog must go. Using a similar approach, it makes the game more challenging if you place large rocks at the top and bottom of the screen. The gecko must then enter an opening in the rock at the top of the screen to finish his trek successfully. This makes sense too, because the openings in the rocks serve as natural hiding places for the gecko.

If you're thinking that the rocks would make good additions to the sprite inventory for Traveling Gecko, then pat yourself on the back! If not, don't feel too bad; it might be because you think they could just be made part of the background. That's true, but there would be a big problem in detecting collisions between the gecko and the rocks. The rocks are there for a reason—to limit the gecko's movement. The only way to limit the gecko's movement is to detect a collision between him and a rock and not let him move if he's colliding with a rock. Without making the rocks sprites, you would have to add a bunch of special case code to a derived `SpriteVector` class to determine whether the gecko is colliding with them. Adding code to a derived `SpriteVector` class isn't the problem, though; the problem is duplicating the collision detection functionality you've already written.

Note

The discussion about the rock sprite brings up a good point in regard to game objects: Practically any graphical object in a game that can be interacted with or handled independently of the background should be implemented as a sprite. Remember that sprites are roughly analogous to cast members in a theatrical play. To carry things a bit further, you can extend the usage of sprites to include the props used in a play. This is essentially the role rocks play in the Traveling Gecko game: props.

The rocks are the last sprites you will need for the game. To summarize what you have thus far, Traveling Gecko requires sprites modeling the following graphical objects:

- Gecko
- Geckocide
- Tarantula
- Rattlesnake
- Gila monster
- Rock

The gecko sprite represents the player and is controlled by the player's user input responses (arrow buttons). The geckocide sprite is used to show a dying gecko and comes in the form of a simple frame animation. The tarantula, rattlesnake, and Gila monster sprites model the predators who are trying to ruin the gecko's trip. Remember that there has to be some method of killing the gecko when he collides with these predators. This is an issue you'll deal with later in this lesson, when you get into writing the code for the MIDlet. Finally, the rock sprite represents the rocks that block the gecko's movement, thereby making it more difficult for him to get across the desert safely.

Game Play

Now that you have an idea of what sprite classes you need to write, let's take a look at the game itself and how it will be played. First, it wouldn't be much fun if the game ended as soon as a predator killed your trusting little gecko. Let's give the player four geckos (lives) to play with—the game isn't over until all four are killed.

Although it is certainly fulfilling to help out a gecko in need, it would also be nice to reward the player with some type of point system. Let's give the player 25 points each time the gecko makes it safely across the desert. Then the player's goodwill for saving a gecko's life is given a numeric value that can be viewed with pride! Because every game ends when all four geckos are killed, you also need to provide the player with a way to start a new game. This is an ideal situation for a command; the player simply selects the `New` command to start a new game.

This finishes the game design for Traveling Gecko. You now have all the information you need to get into the specifics surrounding the MIDlet and its support classes.

Constructing Traveling Gecko

The Traveling Gecko MIDlet is your first complete Java MIDlet game and makes the most of the indispensable sprite classes you developed in Day 17. Rather than guess at what the game will look like before working through the code, it is beneficial to see the game in action so that you understand what you're developing. Figure 18.1 shows the Traveling Gecko MIDlet near the start of a new game.

Traveling Gecko begins by creating the gecko, rocks, and predators. You then use the arrow keys to control the gecko and attempt to guide him safely into the rock opening at the top right of the screen. The score is displayed in the upper-left corner of the screen. Immediately to the right of the score is the number of remaining gecko lives, which are displayed graphically as tiny geckos. The different predators roam across the desert background at different speeds and in different directions, hoping to make a quick meal out of your gecko. If one of them gets lucky, a geckocide object is created to show the dying gecko. The number of remaining lives is then decremented.

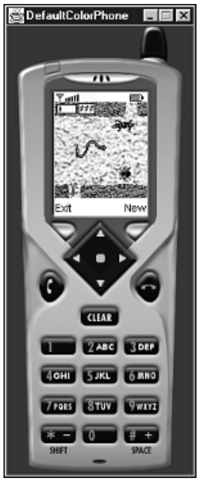


FIGURE 18.1
The Traveling Gecko game is an interesting application of the sprite classes you developed in the preceding lesson.

If you guide the gecko safely across, you receive 25 points and the chance to help him across again. If you're able to help him across a few times, you'll notice that the predators start calling in reinforcements to make things more difficult. If you manage to lose all four of your geckos to the predators, the game ends and you see a message indicating that the game is over. Figure 18.2 shows Traveling Gecko when a game has just ended.

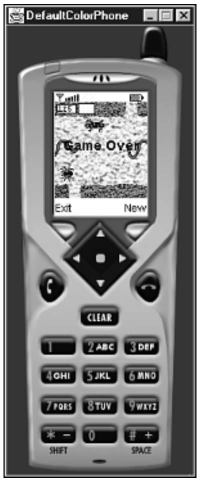


FIGURE 18.2
Even the most skilled Traveling Gecko player will eventually meet the most dreaded of video game phrases, "Game Over."

At this point, if you select the New command, everything starts over. If you haven't checked it out yet, now might be a good time to grab the accompanying CD-ROM and try out the game for yourself. If you can't wait to find out all the details, then by all means skip the CD-ROM and read on.

The Sprite Classes

As you probably guessed, the heart of the Traveling Gecko MIDlet is the extended sprite classes. These classes are all derived from the Sprite class and are responsible for modeling the graphical objects in the Traveling Gecko game, such as the gecko itself and the predators who are after him. The first of these classes is the Gecko class, which represents the gecko that is controlled by the player.

The Gecko Class

The Gecko class is derived from the Sprite class that you developed in the previous lesson. Because the Sprite class is relatively generic in its support for animation, the Gecko class provides a perfect example of how to get very specific functionality out of a derived sprite. The code for the Gecko class clearly reveals how the look and actions of the gecko are implemented in code.

The first place to start with the Gecko class code is the sprite actions that are used to add predators:

```
public static final int SA_ADDGILAMONSTER = 8,
                        SA_ADDRATTLER = 16,
                        SA_ADDTARANTULA = 32;
```

Looking at these sprite actions, it might seem strange to enable the gecko to add predators. However, you'll see in a moment that new predators are added to increase the difficulty of the game. This occurs when the gecko makes it safely across the desert a few times, which can only be detected from within the Gecko class.

The constructor for the Gecko class is pretty simple and is shown in Listing 18.1.

LISTING 18.1 The Gecko Constructor Initializes the Gecko Class

```
public Gecko(int wScr, int hScr) {
  super(image, 0, 1, 0, xStart, yStart, 0, 0, 20, wScr, hScr,
    Sprite.BA_STOP);
  wScreen = wScr;
  hScreen = hScr;
}
```

Notice in the constructor that the `BA_STOP` bounds action is specified, which keeps the gecko from being able to wrap around the sides of the game window. As you see in a moment, the predators won't have this limitation.

The `update()` method in `Gecko` does most of the work of customizing the gecko sprite. Listing 18.2 represents the code for this method.

LISTING 18.2 The `update()` Method Updates the Gecko Sprite

```
public int update() {
  int action = super.update();

  // Toggle the frame and clear the velocity
  if (xVelocity != 0 || yVelocity != 0) {
    frame = 1 - frame;
    setVelocity(0, 0);
  }

  // Has he made it?
  if (yPosition < 8) {
    // Update the score and reposition the gecko
    TGCanvas.score += 25;
    xPosition = xStart;
    yPosition = yStart;

    // See if we should add another bad guy
    if (TGCanvas.score % 100 == 0) {
      Random rand = new Random(Calendar.getInstance().getTime().getTime());
      switch(Math.abs(rand.nextInt() % 3)) {
      case 0:
        // Set flags to add a Gila monster
        action |= Sprite.SA_ADDSPRITE;
        action |= Gecko.SA_ADDGILAMONSTER;
        break;
      case 1:
        // Set flags to add a rattler
        action |= Sprite.SA_ADDSPRITE;
        action |= Gecko.SA_ADDRATTLER;
        break;
      case 2:
        // Set flag to add a tarantula
        action |= Sprite.SA_ADDSPRITE;
        action |= Gecko.SA_ADDTARANTULA;
        break;
      }
    }
  }

  return action;
}
```

The update() method first calls the superclass update() method to handle all the standard sprite updating. It then toggles the gecko's animation frame and clears the velocity. The animation frame is toggled because there are only two frames, 0 and 1. Because there are only two frames, you can just toggle them rather than increment the current frame. The velocity must be cleared because of the way you're handling user input. When the user presses an arrow button to move the gecko, the gecko's velocity is set accordingly, but you want the gecko to move only once for each key press. The solution is to update the gecko, enabling his position to be altered based on the velocity, and then clear the velocity.

The next step in the update() method is to perform a check to see whether the gecko made it across the desert. Because the rocks block him from getting to the top of the screen in all places except the opening, you check his vertical position to see whether he made it. If so, the score is updated, and he is positioned back at the start. Notice that the score is referenced from the TGCanvas class. The score variable, score, is declared as public static in TGCanvas so that other objects can get to it without having access to a TGCanvas object. Technically, this goes against standard object-oriented design practice, but the reality is that it would be very difficult to give access to the score variable using only access methods. You learn about the TGCanvas class a little later in the lesson.

The last chore performed by the update() method is the decision to add a new predator. This determination is based on the score—for every 100 points, a new predator is added. A predator is chosen randomly, and the appropriate sprite action flags are set to trigger the addition of the predator.

Another important method in the Gecko class is addSprite(), which creates the predator sprite objects (see Listing 18.3).

LISTING 18.3 The addSprite() Method Allows the Gecko Sprite to Add Other Sprites Based on a Sprite Action

```
protected Sprite addSprite(int action) {
  // Add new bad guys?
  if ((action & Gecko.SA_ADDGILAMONSTER) == Gecko.SA_ADDGILAMONSTER)
    return new GilaMonster(wScreen, hScreen);
  else if ((action & Gecko.SA_ADDRATTLER) == Gecko.SA_ADDRATTLER)
    return new Rattler(wScreen, hScreen);
  else if ((action & Gecko.SA_ADDTARANTULA) == Gecko.SA_ADDTARANTULA)
    return new Tarantula(wScreen, hScreen);

  return null;
}
```

The `addSprite()` method checks the sprite action flags and creates the appropriate predator. `addSprite()` then returns the newly created sprite so that it can be added to the sprite list.

The last method in the Gecko class is `initResources()`, which is a static helper method used to initialize the sprite images. Listing 18.4 gives the code for this method and shows how the sprite images are loaded.

LISTING 18.4 The `initResources()` Method Initializes the Images for the Gecko Sprite

```
public static void initResources() throws IOException {
  image = new Image[2];
  for (int i = 0; i < 2; i++)
    image[i] = Image.createImage("/Gecko" + i + ".png");
}
```

The `initResources()` method is a standard part of derived sprite classes because it provides a consistent means of loading sprite resources before you actually create a sprite. This makes it possible to initialize all of the sprites in the game from the `TGCanvas` constructor, which you learn about a little later in the lesson.

The `Geckocide` Class

Before getting to the predator classes, let's look at the `Geckocide` class, which is responsible for displaying an animation of a dying gecko. The complete source code for the `Geckocide` class is supplied in Listing 18.5.

LISTING 18.5 The Source Code for the `Geckocide` Class

```
public class Geckocide extends Sprite {
  protected static Image[] image = new Image[4];

  public Geckocide(int xPos, int yPos, int wScreen, int hScreen) {
    super(image, 0, 1, 5, xPos, yPos, 0, 0, 10, wScreen, hScreen,
      Sprite.BA_DIE);
  }

  public static void initResources() throws IOException {
    for (int i = 0; i < 4; i++)
      image[i] = Image.createImage("/Gekcide" + i + ".png");
  }

  public int update() {
    int action = 0;
```

LISTING 18.5 continued

```
      // Die?
      if (frame >= 3) {
        action |= Sprite.SA_KILL;
        return action;
      }

      // Increment the frame
      incFrame();

      return action;
    }
  }
```

I show you the complete code for the `Geckocide` class because it reveals how simple a derived sprite class can be. The `Geckocide` class provides a simple frame-animated sprite that kills itself after one iteration through the frames. This functionality is implemented in the `update()` method, which checks the frame member variable to see whether the animation is finished, and then returns the `SA_KILL` sprite action to have the sprite killed.

The Predator Classes

The predator classes (`Tarantula`, `Rattler`, and `GilaMonster`) are similar to each other and contain relatively little code. The source code for the `Tarantula` class, which provides insight into all the predator classes, is shown in Listing 18.6.

LISTING 18.6 The Source Code for the `Tarantula` Class

```
public class Tarantula extends Sprite {
  private static final int yStart = 67;
  public static Image[] image;

  public Tarantula(int wScreen, int hScreen) {
    super(image, 0, 1, 3, 0, yStart, -1, 0, 30, wScreen, hScreen,
      Sprite.BA_WRAP);
  }

  public static void initResources() throws IOException {
    image = new Image[2];
    for (int i = 0; i < 2; i++)
      image[i] = Image.createImage("/Tarant" + i + ".png");
  }
}
```

The `Tarantula` class uses two images to show a simple animation of a tarantula moving its legs. The `Tarantula()` constructor specifies a fixed horizontal velocity that, when combined with the frame animation, gives the effect of the tarantula creeping across the desert floor. Admittedly, having only two frame animations creates some limitation in how effective the illusion of creeping is in this case. But remember that you're trying to avoid using tons of graphics that take up precious memory and transfer time when loading a MIDlet over the Internet.

The two other predator classes (`Rattler` and `GilaMonster`) are almost identical to `Tarantula`, with the changes being the velocity, horizontal direction, and the images loaded in `initResources()`. Given the code for `Tarantula`, you might wonder why it is even implemented as a derived sprite class. It doesn't really add any new functionality; you could just as easily create a tarantula using the `Sprite` class directly. The truth is that all the predator classes are created as more of a convenience than a necessity. Enabling the classes to initialize their own image resources through `initResources()` and having self-contained constructors that don't take many parameters improve organization.

This goes against typical object-oriented design because the classes technically do not add any new functionality. However, the clean packaging of the classes and their improved ease of use make them justifiable in this case. It might seem a bad idea to break the rules that are so crucial in object-oriented languages such as Java. That's not entirely true. The real skill in object-oriented programming is in knowing when to apply object-oriented techniques and when to use simpler solutions, as you've done here.

The `Rock` Class

The `Rock` class is the last of the `Sprite`-derived classes used in Traveling Gecko, and its code follows in Listing 18.7.

LISTING 18.7 The Source Code for the `Rock` Class

```
public class Rock extends Sprite {
  public static Image[] image;

  public Rock(int i, int wScreen, int hScreen) {
    super(image, i, 0, 0,
      (i % 2 == 0) ? 0 : wScreen - image[i].getWidth(),
      (i < 2) ? 0 : hScreen - image[i].getHeight(),
      0, 0, 40, wScreen, hScreen, Sprite.BA_STOP);
  }
```

LISTING 18.7 continued

```
  public static void initResources() throws IOException {
    image = new Image[4];
    for (int i = 0; i < 4; i++)
      image[i] = Image.createImage("/Rock" + i + ".png");
  }

  public int update() {
    return 0;
  }
}
```

The `Rock` class is similar to the predator classes because it doesn't add too much functionality to `Sprite`. However, there is a very practical reason for creating a `Rock` class instead of just creating rocks as `Sprite` objects. That reason has to do with optimization related to the `update()` method. If you recall, the `update()` method is called for every sprite in the sprite list to enable the animation frame and position to be updated. Rocks have no animation frames, and the positions are fixed. Therefore, you can speed things up a little by overriding `update()` with a "do nothing" version. Because speed is a crucial issue in games, especially MIDlet games, seemingly small optimizations like this can add up in the end.

You learned a great deal about optimizing MIDlet code in Day 12, "Optimizing MIDlet Code." You might have noticed that I haven't followed all the rules presented in Day 12 in the sprite classes and the Traveling Gecko game. The reality is that optimized code isn't always the easiest code to understand because optimizations can sometimes rely on tricky algorithms. For this reason, I presented the examples in this book in a form that is easiest to understand. You can always optimize the code yourself when you reuse it in your own MIDlets.

Note

The trick that is used to help improve speed in the `Rock` class's `update()` method highlights an interesting game programming strategy: Don't be afraid to override unneeded methods with "do nothing" versions. Every bit of execution overhead that you can eliminate will ultimately improve the performance of a game. If you see a way to cut a corner in a derived class simply by overriding a parent class method, do so. Just remember to wait and look for these types of shortcuts after the code is working.

The TGVector Class

The only other sprite-related class to deal with in regard to Traveling Gecko is the derived SpriteVector class, TGVector. The TGVector class is important because it handles the collisions between the sprites. The source code for the TGVector class is shown in Listing 18.8.

LISTING 18.8 The Source Code for the TGVector Class

```
public class TGVector extends SpriteVector {
  private int wScreen, hScreen;

  public TGVector(Background back) {
    super(back);
    wScreen = back.getWidth();
    hScreen = back.getHeight();
  }

  protected boolean collision(int i, int iHit) {
    Sprite s = (Sprite)elementAt(i);
    Sprite sHit = (Sprite)elementAt(iHit);

    // See which sprite is in the collision
    if (sHit.getClass().getName().equals("Rock"))
      // Collided with rock, so stay put
      return true;
    else if (sHit.getClass().getName().equals("Geckocide"))
      // Collided with geckocide, so do nothing
      return false;
    else if (s.getClass().getName().equals("Gecko")) {
      // Kill or reposition it
      int xPos = s.getXPosition();
      int yPos = s.getYPosition();
      if (--TGCanvas.lives <= 0)
        removeElementAt(i--);
      else
        s.setPosition(Gecko.xStart, Gecko.yStart);

      // Collided with bad guy, so add geckocide
      if (add(new Geckocide(xPos, yPos, wScreen, hScreen)) <= i)
        i++;
    }
    return false;
  }
}
```

18

As you can see, the only overridden method in `TGVector` is `collision()`, which is called when a collision occurs between two sprites. The sprite hit in the collision is first checked to see whether it is a `Rock` object. If so, `true` is returned, which causes the sprite that is doing the hitting to stay where it is and not use its updated position. This results in the gecko being stopped when he runs into a rock.

The sprite being hit is then checked to see whether it is a `Geckocide` object, in which case `collision()` returns `false`. This enables the sprite that is doing the hitting to continue on its course and results in a null collision. The purpose of this code is to make sure that `Geckocide` objects don't interfere with any other objects; they are effectively ignored by the collision detection routine.

The real work begins when the hitting sprite is checked to see whether it is a `Gecko` object. If so, you know that the gecko has collided with a predator, so the number of lives is decremented. The `lives` variable is similar to `score` in that it is a `public static` member of the `TGCanvas` class. If `lives` is less than or equal to zero, the game is over and the `Gecko` object is removed from the sprite list. If `lives` is greater than zero, the gecko is repositioned at the starting position. To the player, it appears as if a new gecko has been created, but you're really just moving the old one. Because a gecko has died in either case, a `Geckocide` object is created at the last position of the gecko when it died.

At this point, you've seen all the supporting sprite classes required of Traveling Gecko. The last step is to see what tasks the MIDlet and canvas classes are responsible for.

Inside the Traveling Gecko MIDlet

You learned in Day 17 that the bulk of the animation code for a MIDlet is located in a `Canvas`-derived class, as opposed to the main MIDlet class. This rule holds true in the Traveling Gecko game because the `TGCanvas` class does most of the work in managing the animation for the game. The `TravelingGecko` MIDlet class is left primarily with the job of responding to the `Exit` and `New` commands and of course creating the canvas.

The `TGCanvas` Class

The `TGCanvas` class represents the canvas on which the Traveling Gecko game graphics are drawn. Because the graphics for the game are closely linked to the game play, it makes sense to handle the game play logic in the `TGCanvas` class. For this reason, the majority of the functionality of the Traveling Gecko game is encapsulated in the `TGCanvas` class. Not surprisingly, the `TGCanvas` class in many ways parallels the `AtomsCanvas` class that you saw in the previous lesson. The following list represents all of the member variables for the `TGCanvas` class.

```
private Image       offImage, back, smGecko;
private Graphics    offGrfx;
private TGVector     tgv;
private Gecko        gecko;
protected Timer      animTimer;
private int          animPeriod = 83; // 12 fps
public static int    score, lives;
```

Note

If you're looking on the accompanying CD-ROM for the source code file for the `TGCanvas` class, it is contained within the `TravelingVector.java` source code file. The reason for this is that `TGCanvas` is tightly linked with the `TravelingGecko` MIDlet class and would be very difficult to reuse in a different MIDlet. It makes good organizational sense to place it in the same source code file with the MIDlet.

Most of these member variables should be at least vaguely familiar to you from the `AtomsCanvas` class in the previous lesson. The `Image` member variables represent the offscreen buffer, the background image, and the small geckos used to show the remaining lives near the top of the screen. The `Graphics` member variable, `offGrfx`, stores the graphics context for the offscreen buffer image. The `TGVector` member variable, `tv`, is the customized sprite vector that manages the game sprites. The `animTimer` and `animPeriod` variables control the timing of the game's animation. Finally, the `score` and `lives` member variables hold the current score and number of remaining gecko lives.

The `TGCanvas()` constructor is responsible for getting the game up and running, and it begins by loading all the game images. The code for the `TGCanvas()` constructor is given in Listing 18.9.

LISTING 18.9 The `TGCanvas` Constructor

```
public TGCanvas() {
  // Load the background, small gecko, and sprite images
  try {
    back = Image.createImage("/Back.png");
    smGecko = Image.createImage("/SmGecko.png");
    Gecko.initResources();
    Geckocide.initResources();
    Rock.initResources();
    GilaMonster.initResources();
    Rattler.initResources();
    Tarantula.initResources();
  }
  catch (IOException e) {
```

Listing 18.9 continued

```
    System.err.println("Failed loading images!");
  }

  // Start a new game
  newGame();

  // Create the animation timer and schedule the task
  animTimer = new Timer();
  animTimer.schedule(new AnimationTask(), 0, animPeriod);
}
```

After loading the images for the game, the constructor calls the `newGame()` method to start a new game; you will learn about the `newGame()` method in a moment. Even though the game is now established, not much will take place without creating a timer to run the game animation. Therefore, the `TGCanvas` constructor finishes by creating a `Timer` object and scheduling an animation task through the `schedule()` method. The same `AnimationTask` class that you created in the previous lesson works fine with the Traveling Gecko game because all it does is update and repaint the sprites.

Painting the sprites takes place in the `paint()` method of `TGCanvas`, which is shown in Listing 18.10.

Listing 18.10 The `paint()` Method Handles Painting the Sprites, the Score, the Number of Lives, and the Game Over Message

```
public void paint(Graphics g) {
  // Create the offscreen graphics context
  if (offGrfx == null) {
    offImage = Image.createImage(getWidth(), getHeight());
    offGrfx = offImage.getGraphics();
  }

  // Draw the sprites
  tgv.draw(offGrfx);

  // Draw the score
  offGrfx.setFont(Font.getFont(Font.FACE_MONOSPACE,
    Font.STYLE_PLAIN, Font.SIZE_SMALL));
  offGrfx.drawString(String.valueOf(score), 3, 0,
    Graphics.TOP | Graphics.LEFT);

  // Draw the number of lives
  for (int i = 0; i < (lives - 1); i++)
    offGrfx.drawImage(smGecko, 26 + i * (smGecko.getWidth() + 1),
      3, Graphics.TOP | Graphics.LEFT);
```

LISTING 18.10 continued

```
    // Draw the game over message
    if (lives <= 0) {
      offGrfx.setFont(Font.getFont(Font.FACE_PROPORTIONAL,
        Font.STYLE_BOLD, Font.SIZE_LARGE));
      offGrfx.drawString("Game Over", getWidth() / 2, getHeight() / 2,
        Graphics.HCENTER | Graphics.BASELINE);
    }

    // Draw the offscreen image onto the screen
    g.drawImage(offImage, 0, 0, Graphics.TOP | Graphics.LEFT);
  }
```

The `paint()` method uses the now-familiar double buffering drawing technique to eliminate flicker in the sprite animation. After drawing the sprites with a call to the sprite vector's `draw()` method, the `paint()` method takes care of drawing the score and the number of remaining gecko lives. These two important pieces of information are displayed near the upper-left corner of the screen and are carefully positioned so that they appear over a rectangular section of the first rock sprite. This makes them much easier to see.

After drawing the score and number of lives, the `paint()` method checks the number of lives to see whether the game is over. If so, the phrase "Game Over" is drawn in a large font in the middle of the screen. To keep things interesting, the predator sprites continue to move in the background behind the "Game Over" text.

Earlier in this section I mentioned the `newGame()` method, which is responsible for starting a new game. Listing 18.11 supplies the code for the `newGame()` method.

LISTING 18.11 The `newGame()` Method Starts a New Game

```
public void newGame() {
  // Setup a new game
  lives = 4;
  score = 0;
  tgv = new TGVector(new ImageBackground(back));
  gecko = new Gecko(getWidth(), getHeight());
  tgv.add(gecko);
  for (int i = 0; i < 4; i++)
    tgv.add(new Rock(i, getWidth(), getHeight()));
  tgv.add(new Tarantula(getWidth(), getHeight()));
  tgv.add(new Rattler(getWidth(), getHeight()));
  tgv.add(new GilaMonster(getWidth(), getHeight()));
}
```

The `newGame()` method first initializes the number of lives to four and the score to zero. A new `TGVector` (sprite vector) object is then created, along with a new `Gecko` object.

The Gecko object is then added to the sprite vector, along with four Rock objects. Finally, the method finishes by adding Tarantula, Rattler, and GilaMonster objects to the sprite vector. As you can see, this code sets up everything for a new game. Once the sprites are created and added to the sprite vector, the sprite classes take care of the rest.

The last method in the TGCanvas class is keyPressed(), which is used to respond to the arrow keys. The code for this method is supplied in Listing 18.12.

LISTING 18.12 The keyPressed() Method Responds to Key Presses by Moving the Gecko Sprite

```
public void keyPressed(int keyCode) {
  // Get the game action from the key code
  int action = getGameAction(keyCode);

  // Process the arrow buttons
  switch (action) {
    case LEFT:
      gecko.setVelocity(-3, 0);
      break;

    case RIGHT:
      gecko.setVelocity(3, 0);
      break;

    case UP:
      gecko.setVelocity(0, -3);
      break;

    case DOWN:
      gecko.setVelocity(0, 3);
      break;
    }
}
```

This code is straightforward and calls the getGameAction() method to retrieve the game action associated with the key press. This action is then used in a switch statement to determine whether any of the arrow buttons were pressed on the device. If so, the velocity of the gecko is set accordingly. Keep in mind that earlier in the lesson you saw how the update() code in the Gecko class sets the velocity back to zero each time it is updated. This keeps the gecko from continuing to move upon having its velocity set. The result is that pressing an arrow key moves the gecko a few pixels in a certain direction.

That finishes the code for the TGCanvas class. The last bit of code for the Traveling Gecko MIDlet is the MIDlet class itself, which you learn about next.

The MIDlet Class

As you learned in the previous lesson when you developed the `Atoms` animation MIDlet, most of the animation code for MIDlets that use the sprite classes is contained within a `Canvas`-derived class, not the MIDlet class. This same rule holds true in the Traveling Gecko game, in which case the `TravelingGecko` MIDlet class contains very little animation code, or even game code for that matter. The code for the `TravelingGecko` class, surprisingly similar to the `Atoms` class in Day 17, is in Listing 18.13.

LISTING 18.13 The Complete Source Code for the `TravelingGecko` MIDlet Class

```
public class TravelingGecko extends MIDlet implements CommandListener {
  private Command exitCommand, newCommand;
  private Display display;
  private TGCanvas animScreen;

  public TravelingGecko() {
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit and New commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    newCommand = new Command("New", Command.OK, 2);

    // Create the main animation screen form
    animScreen = new TGCanvas();

    // Set the Exit and New commands
    animScreen.addCommand(exitCommand);
    animScreen.addCommand(newCommand);
    animScreen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the animation screen
    display.setCurrent(animScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
    // Kill the animation timer
    animScreen.animTimer.cancel();
  }
```

LISTING 18.13 continued

```
    public void commandAction(Command c, Displayable s) {
      if (c == exitCommand) {
        destroyApp(false);
        notifyDestroyed();
      }
      else if (c == newCommand) {
        animScreen.newGame();
      }
    }
  }
```

Other than creating a `New` command and then handling the command in the `commandAction()` method, the `TravelingGecko` class is virtually identical to the `Atoms` class from the previous lesson. This reinforces the point made earlier about the bulk of the code for an animation or game being contained within the derived sprite classes and the customized canvas class (`TGCanvas` in this case). Notice in the `TravelingGecko` class that the new game gets started in the `commandAction()` method; the `newGame()` method is called on the canvas member variable, `animScreen`, to start the new game.

You've now seen all the code for the Traveling Gecko MIDlet. I realize that it may seem like a lot of code, but in reality you did most of the work in the preceding lesson when you developed the sprite classes. I encourage you to study this game in detail and make sure that you follow what is happening with the sprites. Then you can try your hand at enhancing it and adding any new features you can dream up. However, before you do that, let's take the game for a spin.

Testing Traveling Gecko

The goal of Traveling Gecko is to guide a little red gecko across a strip of desert to safety between some rocks. You control the gecko with the arrow buttons on a mobile device, or using the arrow keys on the keyboard if you are playing the game in the J2ME emulator. Because this is perhaps the most entertaining MIDlet in the whole book, I encourage you to spend some time playing it and experimenting with the game play. For example, notice how the rocks provide an interesting barrier along the top and bottom of the screen. Also, it is fun to see how close you can get to the predators without triggering a collision detection, and therefore an untimely gecko death.

One interesting aspect of the game is watching how it changes as you progress. A new predator is added to the game each time the gecko travels safely across the desert four times. For every 100 points, you trigger a new predator joining the fray. The new predator is selected at random, so you never know quite how it will affect the game. An extra rattler in some ways is more difficult to deal with because two rattlers take up a lot of space as they move across the screen. Figure 18.3 shows a game of Traveling Gecko almost out of hand with extra predators—the poor gecko really doesn't have much of a chance!

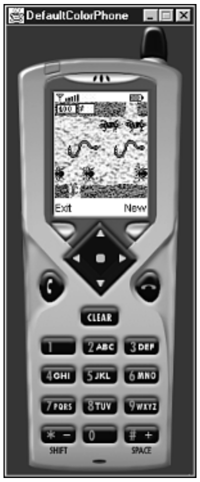


FIGURE 18.3
As you progress through the game and accumulate points, the number of predators increases.

Summary

In this lesson, you developed perhaps the most interesting MIDlet in the book. I am somewhat partial to games, so you might not have found Traveling Gecko as interesting as I do. Traveling Gecko is your first opportunity to apply the sprite classes from Day 17 and in a complete MIDlet game. Although Traveling Gecko is a simple game, it nonetheless is a good starting point for your own MIDlet games. You now have all the functionality required to create MIDlet action games that use both cast- and frame-based animation, as well as interactivity among custom sprites using collision detection and sprite actions.

The next lesson takes a departure from the sprite classes and guides you through the design and development of a Connect4 game that enables you to play against the computer (device). If it isn't bad enough to be beaten in chess by a desktop computer with loads of memory and a high-powered processor, imagine having to admit to your friends that your mobile phone whipped you in Connect4!

Q&A

Q Why derive different sprite classes for all the predators?

A Although it is not necessary to derive different classes for the predators, it is very convenient and makes for good organization because the images used by each predator are linked to the predator class. It is also nice to have constructors with fewer parameters for each predator.

Q How does the `collision()` method in `TGVector` know which types of sprites have collided with each other?

A The `collision()` method uses the name of the sprite class to determine what type of sprite it is. The `getClass()` method gets a `Class` object for the sprite and the `getName()` method gets the class name as a `String` object. When you have the string name of a sprite class (`Gecko`, for example), it is easy to take different actions based on the types of sprites that are colliding.

Q Why are the score and number of lives member variables declared as `public static` in the `TGCanvas` class?

A Because they must be accessible from outside the `TGCanvas` class. More specifically, the `Gecko` class needs to be able to increment the score when the gecko makes it across the screen, and the `TGVector` class needs to be able to decrement the number of lives when the gecko collides with a predator and dies.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you have learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What classic arcade game is Traveling Gecko based on?
2. How is the player rewarded for his kind help in guiding the gecko safely across the hazardous desert?
3. How well would a real gecko fare in the same situation?
4. Why are the rocks in the game modeled as sprites?

Exercises

1. Change the `Gecko` class so that the gecko faces the direction in which he's traveling. Hint: This requires adding new gecko images that show the gecko facing in each direction.
2. Make the destination opening in the rocks vary in position. Hint: Use smaller images for the rocks and more rock sprite objects that can be tiled and rearranged on the screen.
3. Change the predator classes so that the speeds of the predators vary based on the difficulty level (score).
4. Change the predator classes to enable them to travel in either horizontal direction. This could vary with each new game.

DAY 19

The Next Level of MIDlet Gaming

In yesterday's lesson you built your first MIDlet game, Traveling Gecko, which used a set of sprite classes to provide animation. Today you continue with the idea of MIDlet gaming by working through the design and development of another MIDlet game. However, the game you develop in this lesson is quite different from the Traveling Gecko game. In this lesson you develop a strategy game called Connect4 that is based on the popular Connect4 game that is played on an upright stand with checkers. The Connect4 MIDlet in this lesson doesn't require sprite animation, but it does require a smart computer opponent that relies on artificial intelligence (AI) to put up a good fight.

As you develop the Connect4 MIDlet throughout this lesson, you learn about the type of AI strategy used by a game like Connect4, as well as how it is implemented in Java code. By the end of today, you will have created your own worst nightmare: a computer opponent that can consistently beat you in a game of Connect4. You might be thinking that no stupid computer player could ever match wits with you. At this point, I will not make any guarantees, but if any money were involved, mine would be on the player with the silicon brain. The following topics are covered in today's lesson:

- Understanding the fundamentals of artificial intelligence
- Working through the design of the Connect4 game
- Developing the Connect4 game engine classes
- Developing the canvas and MIDlet classes for Connect4
- Testing the Connect4 game

A Crash Course in Game AI

A general definition of artificial intelligence (AI) is computer techniques used to emulate the human thought process. This is a broad definition for AI, as it should be; AI is a very broad research area, and game-related AI is a relatively small subset of the whole of AI knowledge. The primary focus of this lesson is creating a MIDlet game, and therefore I don't want to spend too much time on AI theory. You need just enough AI knowledge to understand how to incorporate AI into the Connect4 game.

NEW TERM *artificial intelligence (AI)*—Techniques used on a computer to emulate the human thought process.

Human thought is not a simple process to emulate, which explains why AI is such a diverse area of research. There are many different approaches to AI, but all of them are an attempt to make human decisions within the limitations of a computer. Most traditional AI systems use a variety of information-based algorithms to make decisions, just as people use a variety of previous experiences and mental rules to make a decision. In the past, the information-based AI algorithms were completely deterministic, meaning that every decision could be traced back to a predictable flow of logic. Figure 19.1 is an example of a purely logical human thought process. Although this purely logical approach to thinking and reacting certainly sounds great, human thinking doesn't usually work this way—if we were all this predictable, it would be quite a boring planet.

Eventually, AI researchers realized that the deterministic approaches to AI were not sufficient to model human thought accurately. Their focus shifted from deterministic AI models to more realistic AI models that attempted to factor in the subtle complexities of human thought, such as best-guess decisions. With people, these types of decisions can result from a combination of past experience, personal bias, or the current state of emotion, in addition to the completely logical decision-making process. Figure 19.2 is an example of this type of thought process. People do not always make scientifically predictable decisions based on analyzing their surroundings and arriving at a logical conclusion.

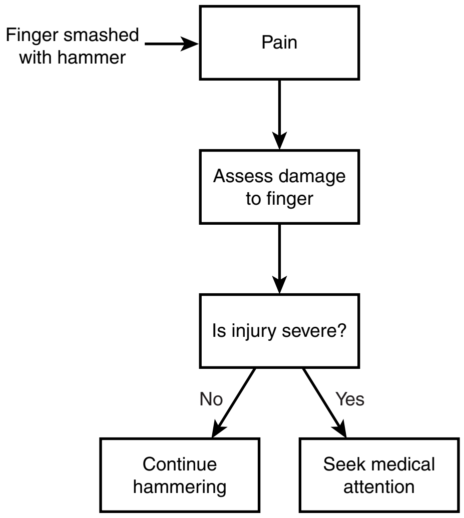


FIGURE 19.1

If we were all completely logical thinkers, the world would probably run much better, but I suspect it wouldn't be as interesting.

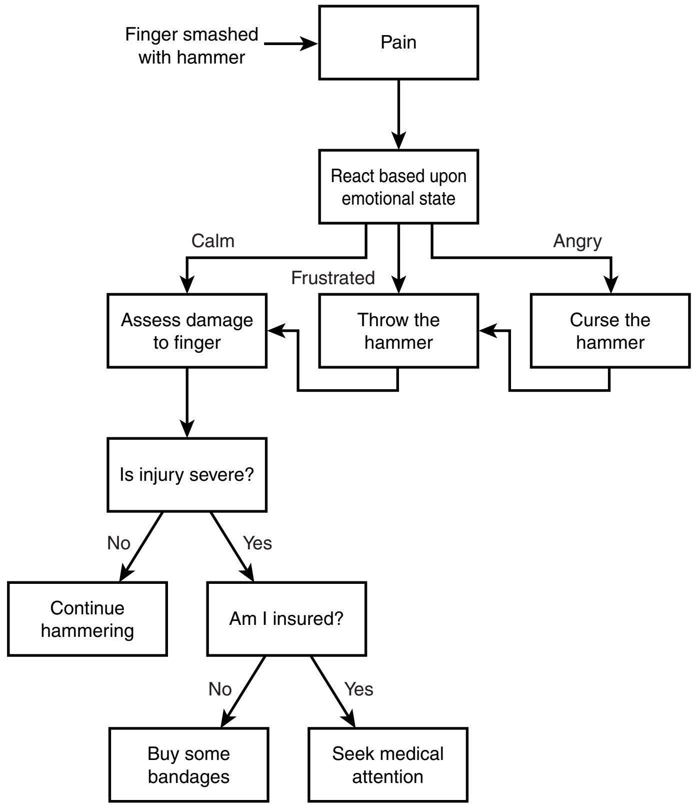


FIGURE 19.2

In reality, the human thought process is more complex than simply applying logic to every situation.

The logic flow in Figure 19.1 is an ideal scenario in which each decision is based on a totally objective logical evaluation of the situation. Figure 19.2 shows a more realistic scenario, which factors in the emotional state of the person, as well as a financial angle (the question of whether the person has insurance). Examining the second scenario from a completely logical angle, it makes no sense for the person to curse at the hammer or throw it, because that only slows down the task at hand. However, this is a completely plausible and fairly common human response to pain and frustration. For an AI carpentry system to model this situation effectively, there would definitely have to be some hammer throwing code in there somewhere!

This hypothetical example gives a clue about how many seemingly unrelated things go into forming a human thought. Likewise, it only makes sense that it should take an extremely complex AI system to model human thought effectively. Most of the time this statement is true. However, the word "effectively" allows a certain degree of interpretation. For our purposes here, effective AI means AI that simulates a worthy computer opponent in a strategy game.

Strategic AI is any AI that is designed to play a game with a fixed set of well-defined rules. For example, a computer-controlled chess player would use strategic AI to determine each move by trying to improve the chances of winning the game. Strategic AI tends to vary more according to the nature of the game because it is so tightly linked to the rules of the game. Even so, there are established and successful approaches to applying strategic AI to many general types of games, such as games played on a rectangular board with pieces. Checkers and chess fit into this group and have a rich history of AI research devoted to them.

New Term *strategic AI*—AI that is designed to play a game with a fixed set of well-defined rules.

Strategic AI, especially for board games, typically involves some form of weighted look-ahead approach to determining the best move to make. The look-ahead is normally used in conjunction with a fixed table of predetermined moves. For a look-ahead to make sense, however, there must be a method of looking at the board at any state and calculating a score. This is known as *weighting* and is often the most difficult part of implementing strategic AI in a board game. As an example of how difficult weighting can be, watch a game of chess or checkers and try to determine who is winning after every single move. Then go a step further and think about trying to calculate a numeric score for each player at each point in the game. Near the end of the game it is easier, but early on it is difficult to tell who is winning because there are so many different things that can happen. Attempting to quantify the state of the game in a numeric score is even more difficult.

New Term *weighting*—A method of looking at a game at any point and calculating a score for each player.

It is possible to successfully calculate a weighted score for strategic games. Using a look-ahead approach with scoring, a strategic AI algorithm can test for every possible move for each player multiple moves into the future and determine which move is the best. This is often referred to as the "least worst" move rather than the best, because the goal typically is to make the move that helps the other player the least, rather than the other way around. Of course, the result is basically the same, but it is an interesting way to look at a game.

Even though a look-ahead approach to implementing strategic AI is useful in many cases, it does have a significant overhead if very much depth is required (in other words, if the computer player must be very smart). This is because the look-ahead depth search suffers from a geometric progression of calculations; that is, the processing overhead significantly increases when the search depth is increased.

To better understand this, consider the case of a computer Backgammon player. The computer player must choose two or four moves from possibly several dozen, as well as decide whether to double or resign. A practical Backgammon program might assign weights to different combinations of positions and calculate the value of each position reachable from the current position and dice roll. A scoring system would then be used to evaluate the worth of each potential position, which gets back to the difficult proposition of scoring, even in a game with simple rules such as Backgammon. Now apply this scenario to a 100-unit war game, with every unit having unique characteristics and the terrain and random factors complicating the issue further. The optimal system of scoring simply cannot be determined in a reasonable amount of time, especially with the limited computing power of a personal computer or, in our case, a mobile device.

The solution in these situations is to settle for a good enough move, rather than the best move. One of the best ways to develop the algorithm for finding the good enough move is to set up the computer to play both sides in a game, using a great deal of variation between the algorithms and weights playing each side. Then let the two computer players battle it out and see which one wins the most. This approach typically involves a lot of tinkering with the AI code, but it can result in very good computer players.

That's enough AI theory for one day. Now that you understand at least some AI fundamentals, we can turn our attention to the design of the Connect4 game.

Designing Connect4

Today's entire focus is on creating a Connect4 game with a computer player that can give a human player at least a good game. After you finish the game, I think you will agree that the computer player can do a lot more than give you a good game. Enough of that for now—you'll have plenty of time to go head-to-head against the computer later.

There are three primary facets to the design of the Connect4 game:

- Game play
- The User interface
- AI strategy

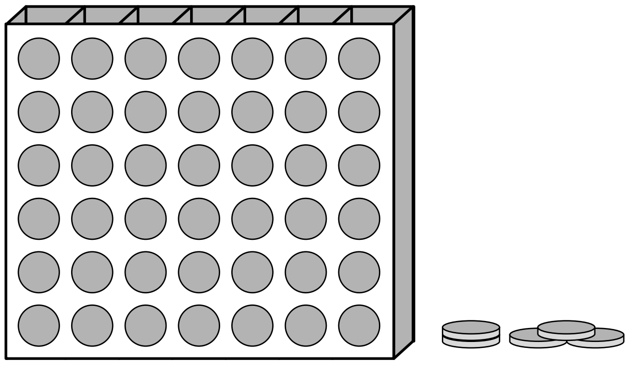
By addressing each of these facets at a design level, you will be better prepared to face the code for the game.

Game Play

If you have never played Connect4, let's go over the rules briefly. It is a simple game similar to Tic-Tac-Toe with goal of completing a continuous row, column, or diagonal series. The game is played on a rectangular board that contains a 7×6 array of positions. Round pieces, similar to checker pieces, represent each move. Figure 19.3 shows what a Connect4 board looks like.

FIGURE 19.3

A Connect4 board provides a vertical playing surface on which you drop circular pieces down slots to make moves within the game.



The main catch to Connect4 is that the board stands upright, with each column being slotted. So a move consists of selecting a column to drop your piece in, and then gravity takes care of the rest. This means that instead of explicitly choosing the vertical position of each move, you can only stack up the pieces vertically. This affects the strategy of the game greatly. The game is won when one of the players manages to connect a horizontal, vertical, or diagonal series of four pieces. There is a chance of a tie, as in Tic-Tac-Toe, but it is less likely because of the number of ways in which the game can be won.

The User Interface

Because the Connect4 game is essentially a board game in which each player takes turns, it isn't necessary to use any animation. Therefore, the sprite classes do not enter the picture with this game. Because you are not going to use animation, the graphics for the game are simplified. This is good because implementing the AI for the game will keep you busy enough. The graphics for the game consist of drawing the board and pieces. Because the computer player is likely to take time calculating its next move, it might also be nice to have a status line. The status line simply indicates the current state of the game, with a message that reads something like “Thinking…” or “Your Turn.”

You are not going to implement animation, so you can add one neat little extra without too much additional work: a column selector arrow. The column selector arrow is shown over the currently selected column so that the human player can tell exactly into which column a piece will be dropped. Each time the player presses the left or right arrow key, the arrow moves to reflect the selected column. The selector arrow is simple to implement, and it gives the game a nice touch. In fact, in some ways the selector arrow is a necessity because the player needs a visual indication of which column a piece will be dropped into.

In addition to using the arrow keys to move the arrow selector, it is necessary to provide a means of exiting the game and starting a new one. This is a perfect opportunity to use a couple of commands: `Exit` and `New`. Adding these commands to the other user interface elements mentioned thus far, we arrive at five user interface requirements for the Connect4 game:

- Commands
- Board
- Pieces
- Status line
- Arrow selector

AI Strategy

The remaining area to cover for Connect4 is the type of AI strategy used by the game and how it affects the general play of the game. If you recall from earlier in the lesson, the most popular type of AI used in games such as Connect4 is strategic AI that makes use of a depth search algorithm linked to a fixed table of rule information. This is exactly the AI strategy you use when developing the Connect4 code, but you will learn the details of that later in the lesson.

For now, consider the AI in terms of the logical components required to make it work. Basically, the AI algorithm should come up with the best possible column for the computer player to drop the next piece into. Keeping that in mind, a good approach would be to break the game into two components: the AI engine that calculates the computer player's next move, and the higher-level MIDlet itself, which uses this move to update the game's visuals.

In this way, the human and computer players' moves are handled very similarly, which makes things much easier to deal with—each move results only in a piece being placed in a position on the game board. Another important benefit of this arrangement is that the AI engine handles all the details of determining whether a move is valid, along with determining whether there is a winner in the game. In this way, the engine is actually more of a general Connect4 game engine rather than just an AI engine.

This approach results in a complete separation of the low-level algorithms necessary to drive the game from the high-level MIDlet issues. However, you still haven't learned anything about the details of the AI itself. How can the computer player know what moves to make and be able to compete with a human player? To answer this question, you must first be able to assess the state of the game from a particular player's perspective. More specifically, you need to be able to calculate a score for a player that determines how he is faring in the game at any given moment.

A player's score reflects his status in the game and how close he is to victory. To calculate the score, you first must be able to know when a game has been won. This might at first seem trivial—just look for a series of four pieces in a row, column, or diagonal, right? That's certainly accurate for you or me, but teaching the computer how to look at the board and determine this isn't quite so simple. One approach is to keep up with every possible combination of piece positions that constitutes a victory. These combinations could be stored in a table and then used to determine how close each player is to winning the game. This sounds fairly cumbersome, but it is not difficult to implement.

With the scoring system in place, you can then use a look-ahead depth search to try out different moves for the computer player and determine the best one. The combination of the table of winning moves and the look-ahead search is what gives the computer player its "intelligence." Although a few more details are involved in implementing the AI for the game, this covers the major points of interest. Let's start writing the code!

Note

Incidentally, a look-ahead depth search is a technique that involves looking into the future of a game and trying out every possible move to determine the best one. A look-ahead depth search is a computer AI version of the common human strategy of planning moves in a game based on what the other player might do in each situation; the primary difference is that the computer is much more efficient and can evaluate many more possibilities.

Constructing Connect4

The Connect4 MIDlet is a complete Connect4 game including strategic AI for a computer player that is good enough to give most people fits. The code for the game will be considerably easier for you to digest if you see the game in action first. Figure 19.4 shows the Connect4 MIDlet in the middle of a game; I happen to be the one losing.

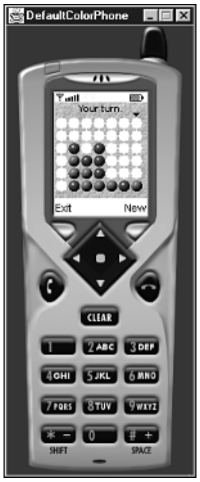


FIGURE 19.4
The Connect4 game demonstrates how to implement basic AI in the context of a MIDlet.

The Connect4 game starts out by setting up an empty game board and giving the human player (you) the first turn. You make a move by pressing the action (fire) button on the device, after which the status message "Thinking…" is displayed while the AI engine determines the computer player's move. Figure 19.5 shows the game while the computer player is "thinking."

Note

Similar to Tic-Tac-Toe, it is technically possible to always win at Connect4 if you are the player to make the first move, regardless of the other player's strategy. Unless you are some kind of Connect4 master, however, this is much tougher to do than you might think.

The computer makes its move and the play shifts back to the human player. This exchange of turns continues until there is a win, loss, or tie. When the game ends, you can start a new one by clicking in the MIDlet window. If you are curious about how well the computer player plays the game, use the accompanying CD-ROM and try out the game for yourself. If not, then let's get busy with the code for the game.

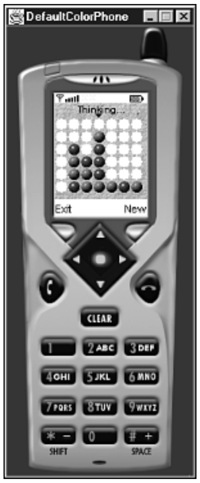


FIGURE 19.5
After you make a move in the Connect4 game, the computer player thinks for a moment and then makes a move.

The Game Engine Classes

The heart of the AI in Connect4 is the Connect4 game engine, which is composed of two classes: `Connect4Engine` and `Connect4State`. The `Connect4Engine` class provides the framework for the game engine itself, and the `Connect4State` class models the current state of a Connect4 game. Together, these two classes provide the functionality necessary to conduct a two-player game of Connect4 with either player being a human or computer player.

These engine classes implement a computer player using look-ahead depth-searching AI; the depth of the search determines the intelligence of the computer player. Deeper searches result in a more intelligent computer player, but at the cost of more processing time.

The `Connect4Engine` Class

The `Connect4Engine` class models the game engine, including the computer player AI, for the Connect4 game; it is based on implementations originally developed by Keith Pomakis and Sven Wiebus. The following member variables are defined in `Connect4Engine`:

```
private static Random rand =
  new Random(Calendar.getInstance().getTime().getTime());
private Connect4State state;
```

The `Connect4Engine` class contains a member variable, `state`, which is of type `Connect4State`. It turns out that `Connect4Engine` delegates much of the dirty work of the AI and game state upkeep to the `Connect4State` class. Even without knowing the details of the `Connect4State` class, you will find that it's not too hard to learn what the class is doing by examining how `Connect4Engine` uses it. Nevertheless, you will learn all the messy details of the `Connect4State` class in a little while.

`Connect4Engine` implements two methods for handling each player's moves: `makeMove()` and `computerMove()`. The code in Listing 19.1 is the source code for the `makeMove()` method.

LISTING 19.1 The `makeMove()` Method Carries Out a Move for the Human Player

```
public int[] makeMove(int player, int xPos) {
  int[] pos = new int[2];

  pos[0] = xPos;
  pos[1] = state.dropPiece(player, xPos);
  return pos;
}
```

The `makeMove()` method is called when the human player makes a move. `makeMove()` takes a player number (`player`), which can be 0 or 1, and a column into which to drop the piece (`xPos`) as parameters. The game state is updated to reflect the move, and the x, y position of the move on the board is returned as an array of two integers (`X` and `Y`). If the move is invalid (for example, if the column is already full), the `Y` value of the position is set to `-1`. Otherwise, it has a value in the range of `0` to `6`.

The `computerMove()` method is called for the computer player to make a move, and its code is shown in Listing 19.2.

LISTING 19.2 The `computerMove()` Method Carries Out a Move for the Computer Player

```
public int[] computerMove(int player, int level) {
  int bestXPos = -1, goodness = 0, bestWorst = -30000;
  int numOfEqual = 0;
  int[] pos = new int[2];

  // Simulate a drop in each of the columns
  for (int i = 0; i < 7; i++) {
    Connect4State tempState = new Connect4State(state);

    // If column is full, move on
    if (tempState.dropPiece(player, i) < 0)
      continue;

    // If this drop wins the game, then we're cool
    if (tempState.isWinner(player)) {
      bestWorst = 25000;
      bestXPos = i;
```

LISTING 19.2 continued

```
      }
      // Otherwise, look ahead to see how good it is
      else
        goodness = tempState.evaluate(player, level, 1, -30000,
          -bestWorst);

      // If this move looks better than previous moves, remember it
      if (goodness > bestWorst) {
        bestWorst = goodness;
        bestXPos = i;
        numOfEqual = 1;
      }

      // If two moves are equally good, make a random choice
      if (goodness == bestWorst) {
        numOfEqual++;
        if (Math.abs(rand.nextInt()) % 10000 <
          (10000 / numOfEqual))
          bestXPos = i;
      }
    }

    // Drop the piece in the best column
    if (bestXPos >= 0) {
      int yPos = state.dropPiece(player, bestXPos);
      if (yPos >= 0) {
        pos[0] = bestXPos;
        pos[1] = yPos;
        return pos;
      }
    }
    return null;
  }
```

The `computerMove()` method handles all the details of determining the best move for the computer player, given the current state of the game. The `level` parameter specifies the depth of the look-ahead search and directly affects both the intelligence of the computer player and the amount of time it takes the computer player to figure out its move. Most of the low-level AI work is handled by the `evaluate()` method in `Connect4State`, which is called by `computerMove()`. Notice that if `computerMove()` determines that there are two equal choices, it randomly chooses one or the other. This gives a little more human feel to the computer player's thought process.

The other three methods in `Connect4Engine` are `getBoard()`, `getWinner()`, and `isTie()`. The code for these is in Listing 19.3.

LISTING 19.3 The `getBoard()`, `isWinner()`, and `isTie()` Methods

```
public int[][] getBoard() {
  return state.board;
}

public boolean isWinner(int player) {
  return state.isWinner(player);
}

public boolean isTie() {
  return state.isTie();
}
```

The `getBoard()` method returns a 7×6 array of integers representing the current state of the game board. The `getWinner()` and `isTie()` methods check the game state member object, `state`, to see whether the game has been won or tied.

That sums up the `Connect4Engine` class. With the exception of the `computerMove()` method, which was admittedly a little tricky, it was pretty painless, don't you think? Brace yourself, because `Connect4State` is a little messier.

The `Connect4State` Class

The `Connect4State` class represents the current state of a game of Connect4. The following are the member variables defined in the `Connect4State` class:

```
public static final int winPlaces = 69, maxPieces = 42,
  Empty = 2;
public static boolean[][][] map;
public int[][]  board = new int[7][6];
public int[][]  score = new int[2][winPlaces];
public int      numPieces;
```

The first step in understanding the `Connect4State` class is to take a look at what the member variables represent. The first three members (`winPlaces`, `maxPieces`, and `Empty`) are static final integers, which means that they are all constant. `winPlaces`, which specifies the number of possible winning combinations on the board, is calculated using the following equation:

```
winPlaces = 4*w*h - 3*w*n - 3*h*n + 3*w + 3*h - 4*n + 2*n*n;
```

This is a general equation that can be applied to any ConnectX-type game. In the equation, `w` and `h` represent the width and height of the game board, and `n` represents the number of pieces that must be in a series to constitute a victory. Because Connect4 uses a 7×6 game board with a four-piece series constituting a win, you can easily calculate `winPlaces` beforehand as 69, which is exactly what is done in `Connect4State`.

The `maxPieces` member specifies the maximum number of pieces that can be placed on the board. It is calculated using the following equation:

```
maxPieces = w*h;
```

This calculation is pretty straightforward. The result is significant because it detects whether there is a tie, which occurs when nobody has won and the number of pieces in the game equals `maxPieces`.

The other constant member, `Empty`, represents an empty space on the board. Each space on the board contains a player number (0 or 1) or the `Empty` constant, which is set to `2`.

The `map` member variable of the `Connect4State` class is a three-dimensional array of Booleans that holds the lookup table of winning combinations. To better understand how the map is laid out, first think of it as a two-dimensional array with the same dimensions as the board for the game; in other words, think of it as a 7×6 two-dimensional array. Now, add the third dimension by attaching an array of winning positions onto each two-dimensional array entry. Each different winning combination in the game is given a unique position within this array (the winning position array is `winPlaces` in length). Each location in this array is set to `true` or `false` based on whether the winning series intersects the associated board position.

Look at an example to see how the map works. Refer to the upper-left space on the game board in Figure 19.3; we will call this position (0,0) on the board. Think about which different winning series of pieces would include this position. Check Figure 19.6 for the answer.

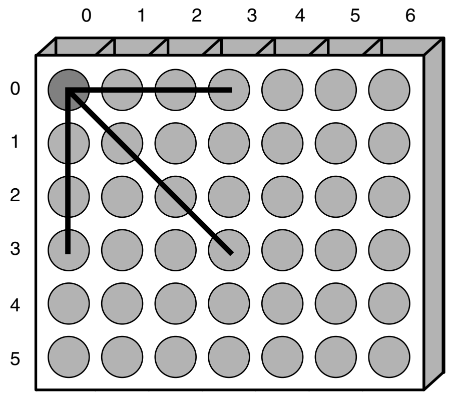


FIGURE 19.6

The winning series possibilities for position (0,0) on the game board are easier to see visually.

As you can see, position (0,0) on the board is a part of three different winning scenarios. Therefore, the winning position array for position (0,0) would have these three entries set to `true`, and all the others would be set to `false`. If the winning moves shown in Figure 19.6 were at positions 11–13 in the winning series array, you would initialize position (0,0) in the map like this:

```
...
map[0][0][9] = false;
map[0][0][10] = false;
map[0][0][11] = true;
map[0][0][12] = true;
map[0][0][13] = true;
map[0][0][14] = false;
map[0][0][15] = false;
...
```

After the entire map is constructed, the AI algorithms can use it to look up winning combinations and determine how close each player is to winning.

Getting back to the `Connect4State` member variables, the `board` member variable is a 7×6 two-dimensional array of integers that represents the state of the game. Each integer entry can be set to either `0` or `1` (for a player occupying that position on the board) or `Empty`.

The `score` member variable contains a two-dimensional array of integers representing the score for the players. The main array in `score` contains a subarray for each player that is `winPlaces` in length. These subarrays contain information describing how close the player is to completing a winning series. Each subarray entry corresponds to a potential winning series and contains a count of how many of the player's pieces occupy the series. If a series is no longer a winning possibility for the player, its entry in the array is set to `0`. Otherwise, the entry is set to 2^p, where `p` is the number of the player's pieces occupying the series. Therefore, if one of these entries is set to `16` (2^4), that player has won.

Rounding out the member variables for `Connect4State` is `numPieces`, which is just a count of how many pieces have been played in the game. `numPieces` is only used to determine whether the game is a tie; in the event of a tie, `numPieces` is equal to `maxPieces`.

That covers the member variables for the `Connect4State` class. You now realize that by understanding the member variables and what they model you already understand a great deal about how the AI works in the game. Let's move on to the methods in `Connect4State`, because there the fun really begins.

The default constructor for `Connect4State` takes on the role of initializing the `map`, `board`, and `score` arrays and consists primarily of a series of nested loops. Listing 19.4 shows the code for this constructor.

LISTING 19.4 The `Connect4State` Default Constructor

```
public Connect4State() {
  // Initialize the map
  int i, j, k, count = 0;
  if (map == null) {
    map = new boolean[7][6][winPlaces];
    for (i = 0; i < 7; i++)
      for (j = 0; j < 6; j++)
        for (k = 0; k < winPlaces; k++)
          map[i][j][k] = false;

    // Set the horizontal win positions
    for (i = 0; i < 6; i++)
      for (j = 0; j < 4; j++) {
        for (k = 0; k < 4; k++)
          map[j + k][i][count] = true;
        count++;
      }

    // Set the vertical win positions
    for (i = 0; i < 7; i++)
      for (j = 0; j < 3; j++) {
        for (k = 0; k < 4; k++)
          map[i][j + k][count] = true;
        count++;
      }

    // Set the forward diagonal win positions
    for (i = 0; i < 3; i++)
      for (j = 0; j < 4; j++) {
        for (k = 0; k < 4; k++)
          map[j + k][i + k][count] = true;
        count++;
      }

    // Set the backward diagonal win positions
    for (i = 0; i < 3; i++)
      for (j = 6; j >= 3; j--) {
        for (k = 0; k < 4; k++)
          map[j - k][i + k][count] = true;
        count++;
      }
  }
```

LISTING 19.4 continued

```
    // Initialize the board
    for (i = 0; i < 7; i++)
      for (j = 0; j < 6; j++)
        board[i][j] = Empty;

    // Initialize the scores
    for (i = 0; i < 2; i++)
      for (j = 0; j < winPlaces; j++)
        score[i][j] = 1;

    numPieces = 0;
  }
```

The default constructor sets the number of pieces to zero. `Connect4State` also has a copy constructor, the source code for which is in Listing 19.5.

LISTING 19.5 The `Connect4State` Copy Constructor

```
  public Connect4State(Connect4State state) {
    // Copy the board
    for (int i = 0; i < 7; i++)
      for (int j = 0; j < 6; j++)
        board[i][j] = state.board[i][j];

    // Copy the scores
    for (int i = 0; i < 2; i++)
      for (int j = 0; j < winPlaces; j++)
        score[i][j] = state.score[i][j];

    numPieces = state.numPieces;
  }
```

C*opy constructors* enable you to create new objects that are copies of existing objects. You need a copy constructor for `Connect4State` because the AI algorithms use temporary state objects a great deal, as you'll see in a moment. The copy constructor for `Connect4State` just copies the contents of each member variable.

NEW TERM *copy constructor*—A constructor that enables you to create new objects that are copies of existing objects.

The `isWinner()` method in `Connect4State` checks to see whether either player has won the game. Listing 19.6 gives the code for this method.

LISTING 19.6 The `isWinner()` Method Determines Whether Anyone Has Won the Game

```
public boolean isWinner(int player) {
  for (int i = 0; i < winPlaces; i++)
    if (score[player][i] == 16)
      return true;
  return false;
}
```

The `isWinner()` method looks for a winner by checking to see whether any member in the `score` array is equal to `16` (2^4). This indicates victory because it means that four pieces occupy the series.

The `isTie()` method checks for a tie by seeing whether `numPieces` equals `maxPieces`, which indicates that the board is full. The source code for `isTie()` follows:

```
public boolean isTie() {
  return (numPieces == maxPieces);
}
```

The `dropPiece()` method handles dropping a new piece onto the board, and its code is shown in Listing 19.7.

LISTING 19.7 The `dropPiece()` Method Drops a Piece onto the Game Board

```
public int dropPiece(int player, int xPos) {
  int yPos = 0;
  while ((board[xPos][yPos] != Empty) && (++yPos < 6))
    ;

  // The column is full
  if (yPos == 6)
    return -1;

  // The move is OK
  board[xPos][yPos] = player;
  numPieces++;
  updateScore(player, xPos, yPos);

  return yPos;
}
```

The `dropPiece()` method takes a player number and an x position (column) as its only parameters. It first checks to make sure room is in the specified column to drop a new piece. Incidentally, you might have noticed from this code that the board is stored upside down in the `board` member variable. The inverted board simplifies the process of adding a new piece. If the move is valid, the entry in the board array is set to `player`, and `numPieces` is incremented. Then the score is updated to reflect the move with a call to `updateScore()`. You'll learn about the `updateScore()` method in a moment.

The `evaluate()` method is where the low-level AI in the game takes place. The source code for the `evaluate()` method is in Listing 19.8.

LISTING 19.8 The `evaluate()` Method Handles the Low-Level Strategy AI of the Game

```
public int evaluate(int player, int level, int depth, int alpha,
  int beta) {
  int goodness, best, maxab = alpha;

  if (level != depth) {
    best = -30000;
    for(int i = 0; i < 7; i++) {
      Connect4State tempState = new Connect4State(this);
      if (tempState.dropPiece(getOtherPlayer(player), i) < 0)
        continue;

      if (tempState.isWinner(getOtherPlayer(player)))
        goodness = 25000 - depth;
      else
        goodness = tempState.evaluate(getOtherPlayer(player),
          level, depth + 1, -beta, -maxab);
      if (goodness > best) {
        best = goodness;
        if (best > maxab)
          maxab = best;
      }
      if (best > beta)
        break;
    }

    // What's good for the other player is bad for this one
    return -best;
  }

  return (calcScore(player) - calcScore(getOtherPlayer(player)));
}
```

The `evaluate()` method's job is to come up with the best move for the computer player, given the parameters for the depth search algorithm. That's a tall order. The algorithm used by `evaluate()` determines the best move based on the calculated value of each possible move. The `alpha` and `beta` parameters specify cutoffs that enable the algorithm to eliminate some moves entirely, thereby speeding things up. It is beyond today's focus to go any further into the low-level theory behind the algorithm. However, if you want to learn more about how it works, feel free to search for artificial intelligence on the Web and read up on common strategy algorithms. A good place to start is the MIT Artificial Intelligence Laboratory, which is located at `http://www.ai.mit.edu/`.

Returning to the Connect4State class, the calcScore() method in Listing 19.9 calculates the score for a player.

Listing 19.9 The calcScore() Method Calculates the Score for a Player

```
private int calcScore(int player) {
  int s = 0;
  for (int i = 0; i < winPlaces; i++)
    s += score[player][i];
  return s;
}
```

In calcScore(), the score of a player is calculated by summing each element in the score array. The updateScore() method shown in Listing 19.10 handles updating the score for a player after a move.

Listing 19.10 The updateScore() Method Updates the Score for a Player

```
private void updateScore(int player, int x, int y) {
  for (int i = 0; i < winPlaces; i++)
    if (map[x][y][i]) {
      score[player][i] <<= 1;
      score[getOtherPlayer(player)][i] = 0;
    }
}
```

The updateScore() method sets the appropriate entries in the score array to reflect the move—the move is specified in the x and y parameters. The last method in Connect4State is getOtherPlayer(), which simply returns the number of the other player:

```
private int getOtherPlayer(int player) {
  return (1 - player);
}
```

That wraps up the game engine. You now have a complete Connect4 game engine with AI support for a computer player. Keep in mind that although you've been thinking in terms of a human versus the computer, the game engine is structured so that you could have any combination of human and computer players.

Inside the Connect4 MIDlet

In the previous two lessons you saw how the bulk of the code for a MIDlet that uses low-level graphics is located in a Canvas-derived class, as opposed to the main MIDlet class. Even though the Connect4 MIDlet doesn't use sprite animation, it still relies on low-level graphics to draw the game board, pieces, arrow selector, and status line. Therefore, the

`Connect4Canvas` class does most of the work managing the game of Connect4. The `Connect4` MIDlet class is left primarily with the job of responding to the `Exit` and `New` commands, not to mention creating the canvas.

The `Connect4Canvas` Class

The `Connect4Canvas` class represents the canvas on which the Connect4 game graphics are drawn. The graphics for the game are obviously linked closely to the game play, which is why it makes sense to handle the game play logic in the `Connect4Canvas` class. In fact, the majority of the code for the game is contained in the `Connect4Canvas` class, which frees up the `Connect4` MIDlet class to manage standard MIDlet tasks such as creating the canvas and handling commands. Following are all of the member variables for the `Connect4Canvas` class:

```
private Image           boardImg, arrowImg;
private Image[]         pieceImg = new Image[2];
private Connect4Engine  gameEngine;
private boolean         gameOver = true;
private int             level = 2, curXPos;
private String          status = new String("Your turn.");
```

The first member variables of interest here are the images that form the basis of the game's graphics. The most important member variable, however, is `gameEngine`, which is a `Connect4Engine` object. There is also a Boolean member variable, `gameOver`, that keeps up with whether the game is over. The `level` member variable specifies the current level of the game and the `curXPos` member keeps up with which column the arrow selector is currently over. Finally, a string member variable stores the current status text that is displayed on the status line. Now that you understand what member variables are involved in the `Connect4Canvas` class, you are ready to delve into its constructor.

Listing 19.11 gives the code for the `Connect4Canvas()` constructor, which loads the game images and starts a new game.

LISTING 19.11 The `Connect4Canvas` Constructor

```
public Connect4Canvas() {
  // Load the board, arrow, and piece images
  try {
    boardImg = Image.createImage("/Board.png");
    arrowImg = Image.createImage("/Arrow.png");
    pieceImg[0] = Image.createImage("/RedPiece.png");
    pieceImg[1] = Image.createImage("/BluePiece.png");
  }
  catch (IOException e) {
    System.err.println("Failed loading images!");
  }
```

LISTING 19.11 continued

```
    // Start a new game
    newGame();
  }
```

The code in the `Connect4Canvas()` constructor is pretty straightforward, especially now that you have some experience with how MIDlet games are initialized. we next look at the `paint()` method, which is much more interesting because it is responsible for the entire graphical appearance of the game. The code for the `paint()` method is in Listing 19.12.

LISTING 19.12 The `paint()` Method Draws the Game Graphics

```
public void paint(Graphics g) {
  // Draw the board
  g.drawImage(boardImg, 0, 0, Graphics.TOP | Graphics.LEFT);

  // Draw the pieces
  int[][] board = gameEngine.getBoard();
  for (int i = 0; i < 7; i++)
    for (int j = 0; j < 6; j++)
      switch(board[i][j]) {
      case 0:
        // Draw a red piece
        g.drawImage(pieceImg[0], (i + 1) * 1 + i *
          pieceImg[0].getWidth(), (6 - j) * 1 + (5 - j) *
          pieceImg[0].getHeight() + 15, Graphics.TOP | Graphics.LEFT);
        break;

      case 1:
        // Draw a blue piece
        g.drawImage(pieceImg[1], (i + 1) * 1 + i *
          pieceImg[1].getWidth(), (6 - j) * 1 + (5 - j) *
          pieceImg[1].getHeight() + 15, Graphics.TOP | Graphics.LEFT);
        break;

      default:
        // Draw an open space on the board
        g.setColor(255, 255, 255);
        g.fillArc((i + 1) * 1 + i *
          pieceImg[0].getWidth(), (6 - j) * 1 + (5 - j) *
          pieceImg[0].getHeight() + 15,
          pieceImg[0].getWidth(),
          pieceImg[0].getHeight(), 0, 360);
        break;
      }
```

LISTING 19.12 continued

```
    // Draw the arrow selector
    if (!gameOver)
      g.drawImage(arrowImg, (curXPos + 1) * 1 + curXPos *
      pieceImg[0].getWidth() + (pieceImg[0].getWidth() -
      arrowImg.getWidth()) / 2, 15 - arrowImg.getHeight(),
      Graphics.TOP | Graphics.LEFT);

    // Draw the game status
    g.setColor(0, 0, 0);
    g.setFont(Font.getFont(Font.FACE_PROPORTIONAL, Font.STYLE_BOLD,
      Font.SIZE_SMALL));
    g.drawString(status, getWidth() / 2, 0,
      Graphics.HCENTER | Graphics.TOP);
  }
```

The `paint()` method draws the board, the pieces, the arrow selector, and the status text. The only tricky part of the `paint()` method is in drawing the pieces in the correct locations, which takes a few calculations. Other than that, the `paint()` method is probably what you expected it to be.

Whereas the `paint()` method is responsible for drawing the game's graphics, the `keyPressed()` method is responsible for handling user interactions through use of the device buttons. The code for the `keyPressed()` method is in Listing 19.13.

LISTING 19.13 The `keyPressed()` Method Handles Key Presses Made by the User

```
public void keyPressed(int keyCode) {
  // Get the game action from the key code
  int action = getGameAction(keyCode);

  // Process the arrow buttons
  switch (action) {
    case LEFT:
      // Move the arrow selector to the left
      curXPos = Math.max(0, curXPos - 1);
      repaint();
      break;

    case RIGHT:
      // Move the arrow selector to the right
      curXPos = Math.min(6, curXPos + 1);
      repaint();
      break;

    case FIRE:
      // Make a move or start a new game (if the game is over)
```

LISTING 19.13 continued

```
        if (gameOver) {
          // Start a new game (or new level)
          newGame();
        }
        else {
          // Make sure the move is valid
          int[] pos = gameEngine.makeMove(0, curXPos);
          if (pos[1] >= 0) {
            if (!gameEngine.isWinner(0))
              if (!gameEngine.isTie()) {
                // Update the display before the computer moves
                status = new String("Thinking...");
                repaint();
                serviceRepaints();

                // Perform the computer move
                pos = gameEngine.computerMove(1, level);
                if (pos[1] >= 0) {
                  if (!gameEngine.isWinner(1))
                    if (!gameEngine.isTie()) {
                      status = new String("Your turn.");
                    }
                    else {
                      status = new String("It's a tie!");
                      gameOver = true;
                    }
                  else {
                    status = new String("You lost!");
                    gameOver = true;
                  }
                }
              }
              else {
                status = new String("It's a tie!");
                gameOver = true;
              }
            else {
              status = new String("You won!");
              level++;
              gameOver = true;
            }
            repaint();
          }
        }
        break;
    }
  }
```

The `keyPressed()` method first checks to see whether the left or right arrow button was pressed. If so, the `curXPos` variable is modified to indicate the new position of the arrow indicator, and the `repaint()` method is called to force a repaint. The code that handles the Fire button is where the `keyPressed()` method really gets interesting. When the Fire button is pressed, it indicates that the player wants to drop a piece in the slot on the board identified by the arrow selector. In other words, you press the Fire button to make a move. In addition to processing the human player's move, the code for the Fire button also carries out the computer player's move.

The first thing the code for the Fire button does is check to see whether the game is over. If so, it starts a new game. This allows the button to double as both a move button and a new game button. If the game is not over, the Fire button code attempts to make the human player's move in the specified column. There is a chance that the move might not be allowed; for example, if the player tries to drop a piece into a full column. If the move is valid, the code checks for a win or tie and then updates the status text. This is where a neat little trick is played to ensure that the screen is properly updated.

If you look ahead in the code, you will see that the computer player's move is processed just after the human player's move. Normally, a call to the `repaint()` method won't result in the screen being repainted until the code making the call finishes. In this case it means that the repaint won't occur until the `keyPressed()` method finishes executing, which is a bad thing because you want the human player's move to appear while the computer player is thinking. This problem is remedied with the call to the `serviceRepaints()` method, which forces a repaint immediately. The `serviceRepaints()` method is a handy little method that serves as a lifesaver for this particular problem.

Moving right along in the Fire button code, the computer player's turn is carried out by calling the `computerMove()` method on the `gameEngine` member variable. If the move is successful, the code checks for a win or tie and then updates the status text. After the move has been made, the code to check for a win or tie is virtually the same for the human and computer players.

Notice throughout the Fire button code that if the human player has won, the level of the game is increased. This is significant because it means that the game gets progressively more difficult as you win. Along with making the game more interesting, this subtlety is likely to make the game somewhat addictive because it takes only a couple of level increases before the computer becomes very tough to beat.

Note

The `level` member variable determines how smart the computer player is by affecting the depth of the look-ahead search. This is carried out by the level being passed as the second parameter of the `computerMove()` method. If you find the game too easy or too difficult, feel free to tinker with the `level` member variable or even supply your own calculation as the second parameter to `computerMove()`.

The last method in the `Connect4Canvas` class is `newGame()`, which (surprise!) sets up a new game:

```
public void newGame() {
  // Setup a new game
  gameEngine = new Connect4Engine();
  gameOver = false;
  status = new String("Your turn.");
  repaint();
}
```

The `newGame()` method starts a new game by re-creating the game engine and reinitializing all the game status member variables except for `level`. This is important because it results in the level increasing after each win by the human player. The only drawback is that the computer player plays somewhat slower (due to the increased depth search) with each increasing level. On the other hand, the computer player gets much smarter with each human player victory, so don't be surprised if the computer starts outsmarting you after a few games.

You are now very close to having the complete Connect4 MIDlet developed. All that is left is the `MIDlet` class, which is fortunately pretty lean.

The MIDlet Class

Judging by how much work was carried out by the `Connect4Canvas` class, you probably have a hunch that the code for `Connect4` MIDlet class is relatively simple. The code for the `Connect4` class is shown in Listing 19.14, which proves that your hunch is entirely accurate.

LISTING 19.14 The Complete Source Code for the `Connect4` MIDlet Class

```
public class Connect4 extends MIDlet implements CommandListener {
  private Command exitCommand, newCommand;
  private Display display;
  private Connect4Canvas screen;

  public Connect4() {
```

LISTING 19.14 continued

```
    // Get the Display object for the MIDlet
    display = Display.getDisplay(this);

    // Create the Exit and New commands
    exitCommand = new Command("Exit", Command.EXIT, 2);
    newCommand = new Command("New", Command.OK, 2);

    // Create the main screen form
    screen = new Connect4Canvas();

    // Set the Exit and New commands
    screen.addCommand(exitCommand);
    screen.addCommand(newCommand);
    screen.setCommandListener(this);
  }

  public void startApp() throws MIDletStateChangeException {
    // Set the current display to the screen
    display.setCurrent(screen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == newCommand) {
      screen.newGame();
    }
  }
}
```

Not surprisingly, this code is about as generic as a MIDlet can get. Other than creating the `Exit` and `New` commands and creating and setting the `Connect4Canvas` object, this code could appear in almost any MIDlet. The reality is that we have delegated the details of the Connect4 game to the `Connect4Canvas` class, as well as the game engine classes.

This wraps up the development of the Connect4 MIDlet. I will be the first to admit that the AI code for the game is a little tricky, but that is the nature of AI programming. Short of spending half the book on AI theory, I opted to present the code and have you study it in as much detail as you want. The bottom line is that the AI code makes it possible to build a very interesting and highly playable MIDlet strategy game.

Testing Connect4

You have already seen the Connect4 MIDlet in action earlier in the lesson. If you did not stop to try out the MIDlet earlier in the lesson, take the time to do so now. This MIDlet is particularly interesting because the AI of the computer player enables you to experiment with different strategies to see how the computer responds. The computer player has a particular weakness that enables you to beat it consistently on the first two levels. However, after you get two victories under your belt, you will find that the computer player suddenly gets very tough to beat.

Maybe you are smarter than I am and will breeze through several more levels before you run into trouble, and if so you can congratulate yourself on being a better Connect4 player than this geeky author! It's kind of scary to create a computer program that is fully capable of outsmarting you. Many of us imagine such programs developed on monstrous mainframe computers by secretive Defense Department computer scientists. However, the Connect4 MIDlet demonstrates how a device as relatively low-powered as a mobile phone harnesses enough processing power to outsmart the majority of us.

Summary

In today's lesson, you learned how to move beyond simple action games to create a strategy game complete with artificial intelligence (AI). The game targeted in this lesson is the popular Connect4 game that many of us played as kids. The lesson began by presenting a brief AI primer to get the basics of AI under your belt. From there, you began designing the Connect4 game and assessing what it requires in terms of functional components. You then quickly jumped into the development of the Connect4 MIDlet, starting with a couple of classes that make up the game engine for the game. The lesson concluded by leading you through the remainder of the MIDlet's development, as well as testing the game.

This lesson concludes this section of the book, which focused on entertainment MIDlets. Although we all love to be entertained, it would get old quickly if we were entertained all the time. For this reason, we are going to shift gears and enter the final section of the book, exploring the various J2ME wireless devices that are currently available.

Q&A

Q Could a similar AI approach be used in another board game, such as Checkers?

A Yes, but the technique of calculating a score for each player would be different, because winning Checkers is based on jumping pieces instead of lining them up in a row.

Q Is this the best AI approach to use for implementing the Connect4 computer player?

A Probably not, but it depends on how you define “best.” In terms of simplicity, this approach might well be one of the best. In my research, I ran across a few other Connect4 AI strategies, but I settled on this one because it is relatively easy to implement and understand. There is no doubt that smarter AI strategies exist, but they are almost certainly more difficult to implement. Keep in mind, however, that most other AI strategies in games like Connect4 still use a conceptually similar approach involving look-ahead depth searching. The primary difference usually lies in how the scores are calculated.

Q Will the Connect4 engine work with a larger board or a different number of pieces to connect in a series?

A Absolutely. You just need to alter the size of the game state arrays as well as all the loops that work with them, along with recalculating the values of the `winPlaces` and `maxPieces` member variables in the `Connect4State` class.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you’ve learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, “Quiz Answers.”

Quiz

1. With respect to artificial intelligence, what is weighting?
2. What is the purpose of the map in the AI for the game?
3. How does the `level` member variable affect the AI for the computer player?
4. What is the significance of the `serviceRepaints()` method that is called in the Fire button code in the `keyPressed()` method?

Exercises

1. Integrate the sprite classes into the Connect4 MIDlet to animate dropping the pieces.
2. Modify Connect4 so that two computer players battle it out. Hint: You can use a timer to slow down the moves so that you can watch the game unfold.
3. In the computer-versus-computer version of Connect4 you just created, try out different values for the level of each player, and watch the results.

Part V

Exploring J2ME Wireless Technologies

20 A J2ME Wireless Tour

21 The Future of J2ME

DAY 20

A J2ME Wireless Tour

Throughout the book thus far you've focused on the specifics of the CLDC and MIDP APIs and what they offer in terms of developing MIDlets for wireless mobile devices that support J2ME. You have not learned exactly what devices support J2ME and what devices you might want to specifically target as you begin constructing MIDlets of your own. Additionally, it is worth evaluating other J2ME applications that are currently available; you certainly wouldn't want to re-invent the wheel unnecessarily. Having a solid grasp on the J2ME devices and applications currently in circulation at the time of this writing (May 2001) will give you a better feel for where you might focus your own J2ME development efforts.

This lesson provides a brief assessment of the J2ME wireless landscape and introduces you to the different J2ME devices and applications that are either currently available or soon to be released. The following major topics are covered in this lesson:

- Getting acquainted with J2ME devices
- Understanding the different types of J2ME applications
- Taking stock of the current J2ME application offerings

Exploring J2ME Devices

As you know, the premise behind J2ME is the support of Java on compact devices as opposed to desktop or laptop computers. More specifically, the CLDC specification describes a class of "connected limited" devices, which have access to the Internet and are constrained with respect to memory, processing power, and other system resources such as screen size. The MIDP specification takes things a step further by describing a class of devices that have wireless Internet access. The classification of MIDP devices might seem somewhat narrow, but you will find that it actually encompasses a wide range of devices. More specifically, the following types of devices fall under the MIDP device classification:

- Mobile phones
- Pagers
- Handheld computers

These different devices offer a significant range of system features, but MIDlets are capable of being executed across all three types. Perhaps more interesting, the same executable MIDlet bytecode can be executed across these platforms, thanks to the inherent cross-platform nature of Java. This is an extremely important benefit of J2ME in mobile devices because these devices typically utilize a wide range of operating systems, which is in stark contrast to desktop computers that rely on only a handful of operating systems.

Note

Keep in mind that a device capable of running MIDlets is basically a device with a Java virtual machine that supports the MIDP specification.

Now that you have an idea of the different types of devices that are capable of running MIDlets, let's look a little closer at some specific devices that are available (or nearly available) for commercial use.

Java-Powered Mobile Phones

Perhaps the most anticipated devices to support J2ME are mobile phones, which have gained in popularity tremendously over the past few years. Many people carry their mobile phones with them as if they were an extension of their bodies, and as much as some people talk on them, they might even look like an extension of their bodies! In all seriousness, mobile phones represent a technology that began as a neat gadget, and has now become something that many of us rely on as an everyday convenience. This is often the way new technologies enter our lives; think back to video cameras, VCRs, cordless phones, and video games. The progression of devices that start as gadgets and evolve into everyday conveniences, or even necessities, is common.

The addition of J2ME to the mobile phone equation has significant ramifications when it comes to the usefulness of mobile phones. Up until now phones have basically been used as a means to communicate by voice with other people from virtually any location. One of the only significant innovations in mobile phones (prior to J2ME) is the two-way radio feature present in some Motorola phones. If you've never used such a phone, you can press a button and immediately talk live with someone who owns a similar phone. In other words, you don't have to make a call—you can communicate using the phone just as if it were a two-way radio. These phones also support traditional mobile phone calls.

Note

Motorola also popularized portable two-way radios recently with their line of TalkAbout radios. These devices are not phones, and currently aren't a part of the Java equation, but they are interesting devices nonetheless. The latest TalkAbout radios are actually text-based message devices that enable you to send short text messages to other TalkAbout users.

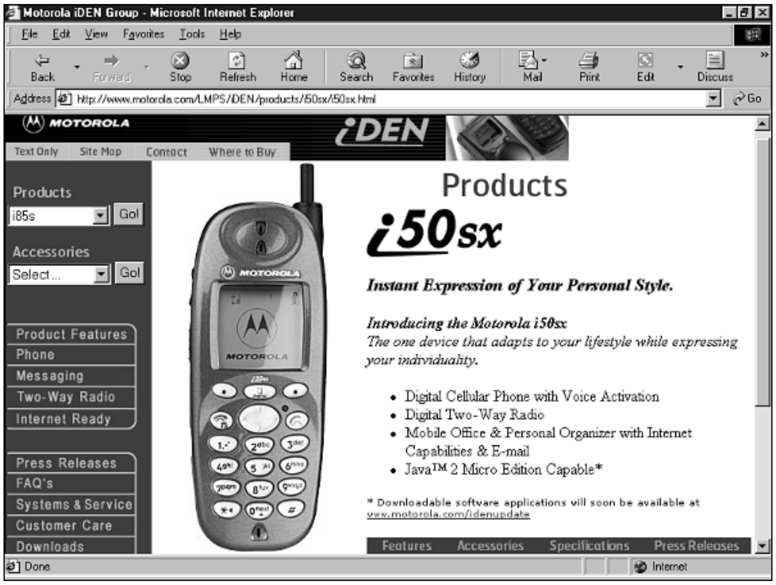
Motorola's two-way radio feature is important because it highlights the significance of introducing J2ME to mobile phones. J2ME is positioned to quite literally reshape the manner in which we all perceive mobile communications. This is relatively easy to see when you consider that a Java-powered mobile phone has two unique attributes that existing mobile communication devices do not have: an Internet connection and the capability to run Java programs. This opens up all kinds of interesting possibilities for developing wireless Java programs (MIDlets). Of course, you've already learned about many of these possibilities throughout the book thus far. Next we will explore some real Java phones that are available today.

Motorola has positioned itself to be the lead player in the Java mobile phone revolution. More specifically, Motorola is the first mobile phone vendor to offer Java-enabled phones based on J2ME. These phones are part of Motorola's iDEN product line and are currently available. As of this writing, Motorola's iDEN phones that support J2ME are only available in the United States, but there is every reason to expect international support in the near future. In addition to being the first vendor to hit the street with real, functioning Java phones, Motorola also offers a significant support system for users of the phones and developers of phone applications (MIDlets).

Motorola's entry into the Java phone market consists of the following two models: i85s and i50sx. Both of these phones include the Java K virtual machine (KVM), as well as full support for the CLDC and MIDP APIs. The i85s phone is geared toward business professionals, whereas the i50sx is targeted at a broader range of phone users who demand personalization, such as colorful faceplates. Figure 20.1 shows the i50sx phone, as seen on Motorola's iDEN Web site (`http://www.motorola.com/iden/`).

FIGURE 20.1

The Motorola i50sx mobile phone is one of the first phones to include full support for J2ME, which enables it to run Java MIDlets.


Motorola iDEN Group - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.motorola.com/LMPS/iDEN/products/i50sx/i50sx.html
Go
MOTOROLA
iDEN
Text Only Site Map Contact Where to Buy
Products
i85s Go!
Accessories
Select... Go!
Product Features
Phone
Messaging
Two-Way Radio
Internet Ready
Press Releases
FAQ's
Systems & Service
Customer Care
Downloads
Features Accessories Specifications Press Releases
Done
Internet

Products

i50sx

Instant Expression of Your Personal Style.

Introducing the Motorola i50sx
The one device that adapts to your lifestyle while expressing your individuality.

- Digital Cellular Phone with Voice Activation
- Digital Two-Way Radio
- Mobile Office & Personal Organizer with Internet Capabilities & E-mail
- Java™ 2 Micro Edition Capable*

* Downloadable software applications will soon be available at www.motorola.com/idenupdate

Both of the Motorola Java phones have enough memory to store several MIDlets at once, which is obviously to be expected given what you have learned about J2ME throughout the book. Several MIDlets come with each phone, including personal productivity applications as well as a puzzle game called Borkov that was developed for Motorola exclusively by Sega. These phones represent a major step toward the convergence of pure communication devices, personal digital assistants, and handheld computers.

The Motorola i85s and i50sx phones also support several optional accessories, some of which are somewhat new to mobile phones. These include chargers, holders, a hands-free automobile adapter, and a foldable keyboard that dramatically improves the entry of text data. Your experience with the J2ME emulator should be proof enough that a keyboard could prove very useful when working with some MIDlets.

Note

This mobile phone discussion is admittedly Motorola-centric because Motorola is the first mobile phone vendor to ship Java-powered phones. Ericsson, Nokia, and other phone vendors will release comparable phones soon. The primary benefit of Java-powered phones to J2ME developers is that MIDlets will run on any of the phones because of the cross-platform nature of Java.

Finding and installing software for Java-powered phones is an important issue. Motorola phones include several standard MIDlets, but users will want to install additional MIDlets to carry out specific productivity tasks or maybe play other types of games. To facilitate this process, Motorola has created the iDEN Update Web site, located at `http://www.motorola.com/idenupdate/`. This Web site provides a place for iDEN phone users to locate and purchase MIDlets for use in their Java-powered phones. The Web site provides the necessary framework that enables users to purchase and immediately install MIDlets.

Note

Another interesting service that Motorola offers is a support program for MIDlet developers, which is intended to help bolster the development of MIDlets. This program assists MIDlet developers in developing, marketing, and distributing their MIDlets to Motorola iDEN phone users. iDEN is Motorola's wireless digital network that enables mobile phones to be used as two-way radios, to send and receive test messages, to communicate over the Internet, and to allow traditional voice communications. For more information, check out the iDEN Developer Web site at `http://www.motorola.com/idendev/`.

Pagers That Speak Java

Pagers are another type of mobile communications device that is entering the realm of Java. These are sophisticated, interactive pagers that support e-mail text messaging. One of the heavy hitters in the interactive pager marketplace is Research In Motion (RIM), which makes several pagers that support wireless e-mail. Currently, business professionals are the primary users of these pagers.

Note

Several devices already exist that enable wireless text messaging, but RIM devices are probably the most powerful in terms of providing full support for e-mail messaging.

I mention RIM devices not just because they are currently some of the most popular messaging devices, but because RIM is one of the first interactive pager vendors to embrace Java in their devices. As of this writing, none of the RIM devices support Java yet, but the Java-enabled devices could very well be available by the time you read this. RIM makes several different devices, all of which have keyboards to enable the entry of text messages. Although the keyboards aren't exactly as convenient as a full-size computer keyboard, you'd be surprised at how fast you can type on them after you get the hang of it. My wife is a RIM user, and she is quite the typing demon when entering text messages using the small keyboard.

Figure 20.2 shows two RIM interactive paging devices, which will soon support J2ME. For more information on these devices, visit the RIM Web site at `http://www.rim.net/`.

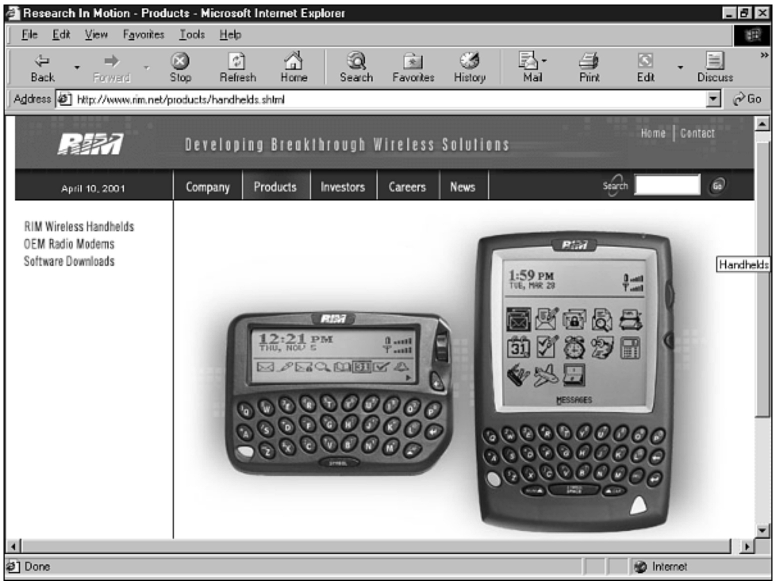

Research In Motion - Products - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.rim.net/products/handhelds.shtml
Go
RIM
Developing Breakthrough Wireless Solutions
Home | Contact
April 10, 2001
Company Products Investors Careers News
Search
RIM Wireless Handhelds
OEM Radio Modems
Software Downloads
Handhelds
Done
Internet

FIGURE 20.2
The RIM interactive pagers currently support e-mail messaging, and will very soon support Java and enable you to run MIDlets.

As you can see in Figure 20.2, RIM devices open up some interesting options for MIDlet development because they have larger screens than most mobile phones. The larger RIM device has a screen that rivals full-blown handheld computers, thus blurring the line between pagers and handheld computers. It will be very interesting to see what kinds of J2ME applications appear for these devices.

Note

If you take a closer look at RIM devices you might wonder what Blackberry is. Blackberry is the name of RIM's wireless e-mail solution that includes special e-mail software coupled with a RIM device. For this reason, RIM devices are sometimes referred to as *Blackberry handhelds*. Many people have found themselves addicted to wireless messaging using these devices; therefore some wireless industry insiders refer to these devices as "crackberries."

Handheld Java Computers

The last type of device of interest to you as a J2ME developer is handheld computers, which are now embracing the wireless Java revolution. The most popular handheld computer at the moment is the Palm line of personal digital assistants (PDAs). Palm handhelds have become enormously popular over the past few years, along with similar Palm-compatible devices made by a company called Handspring. You could argue that a personal digital assistant (such as a Palm device) doesn't really qualify as a full-blown handheld computer. This argument has some validity when you compare the features of a Palm device with those of a Pocket PC device. Pocket PCs are true handheld computers that run a special mobile version of Windows called Pocket PC.

I could debate the technical nuances of why a Pocket PC is a legitimate handheld computer and a Palm is not, but the reality is that both are used to perform many tasks typically carried out on a desktop or laptop computer. Yes, Pocket PCs are considerably more powerful than Palm devices, but both are extremely powerful handheld computing devices that fit in the palm of your hand. With that bit of business out of the way, let's talk about how Java fits into the world of handheld computing.

Java's first entry into handheld computing comes by way of Palm devices. The CLDC and MIDP APIs are now supported on Palm devices, thanks to an effort by Sun to extend J2ME to the Palm platform. J2ME runs on Palm devices thanks to the K virtual machine (KVM). In addition to the CLDC and MIDP APIs, which are supported on the Palm platform today, Sun is working with Palm to develop an API specifically targeted at PDAs. This will make it possible to develop J2ME applications that integrate more closely with the existing features common in most PDAs, such as a contact list, meeting scheduler, and task manager.

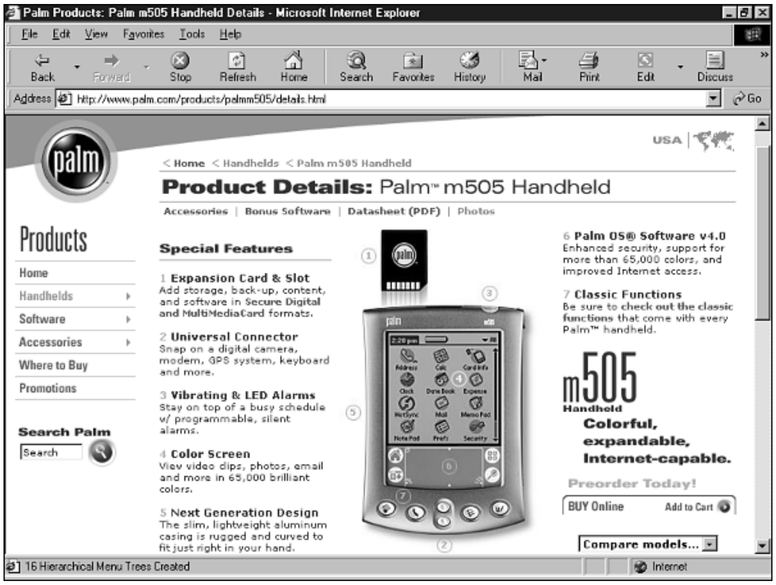
Perhaps the most interesting facet of the integration of Java into Palm devices is that it is a marriage of a significant developer base with a very large user base. Many Java developers are itching to work with J2ME, and even more Palm users would love to see more interesting applications emerge for their devices. In this regard, the combination of Java and Palm devices will quite likely be a win-win situation for everyone involved.

Note

One concern with respect to Java and Palm devices is that the Palm operating system doesn't currently include Java support automatically. This means that the user must somehow install the Java KVM to run MIDlets on a Palm device. Fortunately, Sun is making available a special Palm KVM that can be included with J2ME applications that target the Palm platform.

Now that you have a basic understanding of how Java is being integrated with handheld computers, it's worth taking a quick look at a few of the latest Palm devices that you, as a J2ME developer, might consider using. Keep in mind that all Palm devices are compatible with each other, and therefore all Palm devices are capable of supporting Java. Even so, you might have more fun tinkering with a higher end device with a color screen. Ultimately, this decision is up to you.

Figure 20.3 shows the Palm m505 handheld computer, which is one of the latest Palm devices as of this writing. This device includes a color screen and an expandable storage slot. To learn more about the m505 device, as well as other Palm devices, visit the Palm Web site at `http://www.palm.com/`.


Palm Products: Palm m505 Handheld Details - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.palm.com/products/palmm505/details.html
Go
palm
USA
Products
Home
Handhelds
Software
Accessories
Where to Buy
Promotions
Search Palm
Search
Preorder Today!
BUY Online
Add to Cart
Compare models...
16 Hierarchical Menu Trees Created
Internet

< Home < Handhelds < Palm m505 Handheld

Product Details: Palm™ m505 Handheld

Accessories | Bonus Software | Datasheet (PDF) | Photos

Special Features

1 **Expansion Card & Slot**
Add storage, back-up, content, and software in Secure Digital and MultiMediaCard formats.

2 **Universal Connector**
Snap on a digital camera, modem, GPS system, keyboard and more.

3 **Vibrating & LED Alarms**
Stay on top of a busy schedule w/ programmable, silent alarms.

4 **Color Screen**
View video clips, photos, email and more in 65,000 brilliant colors.

5 **Next Generation Design**
The slim, lightweight aluminum casing is rugged and curved to fit just right in your hand.

6 **Palm OS® Software v4.0**
Enhanced security, support for more than 65,000 colors, and improved Internet access.

7 **Classic Functions**
Be sure to check out the classic functions that come with every Palm™ handheld.

m505 Handheld

Colorful, expandable, Internet-capable.

FIGURE 20.3
The Palm m505 handheld computer is an excellent example of a high-end device that is capable of running MIDlets.

Palm has seen much success with their handheld devices, but they have been challenged in a big way by a competitor named Handspring that makes a popular line of Palm-compatible devices. The Handspring Visor is a Palm-compatible device that in many ways upped the ante on Palm by supporting an expandable slot (known as a Springboard) long before Palm did. This enabled Visor users to purchase add-on modules such as modems and software suites, and plug them directly into their devices. Visor handhelds were also the first handhelds to come in cute colors, which went a long way toward selling them to individuals who appreciated being able to add a personal touch to their device.

In terms of Java, the Handspring Visor handhelds are not much different from Palm handhelds. In fact, their similarities make it possible to run Java MIDlets on devices from either manufacturer. Figure 20.4 shows the popular Visor Edge handheld, which is one of the latest offerings from Handspring as of this writing.

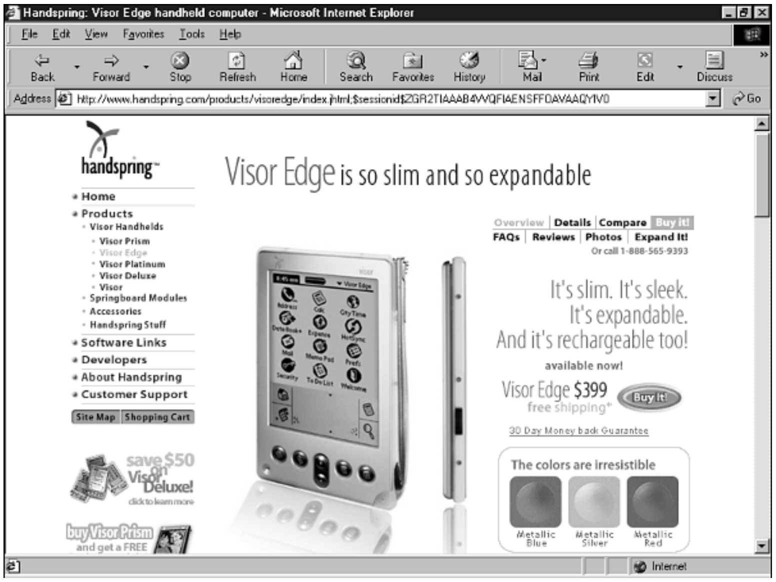

Handspring: Visor Edge handheld computer - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.handspring.com/products/visoredge/index.jhtml;$sessionid$ZGR2TIAAAB4VVQFIAENSFFQAVAAQYIV0
Go
handspring
Home
Products
Visor Handhelds
Visor Prism
Visor Edge
Visor Platinum
Visor Deluxe
Visor
Springboard Modules
Accessories
Handspring Stuff
Software Links
Developers
About Handspring
Customer Support
Site Map
Shopping Cart
Visor Edge is so slim and so expandable
Overview Details Compare Buy it!
FAQs Reviews Photos Expand It!
Or call 1-888-565-9393
It's slim. It's sleek.
It's expandable.
And it's rechargeable too!
available now!
Visor Edge $399
free shipping*
Buy It!
30 Day Money back Guarantee
The colors are irresistible
Metallic Blue
Metallic Silver
Metallic Red
save $50 on Visor Deluxe!
buy Visor Prism and get a FREE
Internet

FIGURE 20.4

The Handspring Visor Edge handheld computer is a Palm-compatible device that is also capable of running MIDlets, thanks to J2ME.

Both Palm and Handspring devices represent a significant leap forward for J2ME by giving J2ME developers a broad range of users to target with their MIDlets.

Note

Earlier in this section I compared Palm devices and Pocket PCs. You might be surprised that I didn't mention Pocket PCs as currently offering J2ME support. That's because right now none do. Even so, I expect to see J2ME married with the Pocket PC platform in the very near future. However, given the power of Pocket PC devices, I wouldn't be surprised to see them targeted by the CDC (Connected Device Configuration), as opposed to the CLDC. This will likely mean that a different profile (other than MIDP) will be required to develop J2ME applications for Pocket PCs. You learn more about the CDC in the next lesson, "The Future of J2ME."

Assessing the J2ME Application Landscape

In addition to J2ME devices, it's worth it as a J2ME developer to stay on top of current developments with J2ME applications. By understanding what's out there in the marketplace, you can better focus your efforts and possibly fill niches in the J2ME application landscape. Maybe you're a serious J2ME device user, and just want to know what kinds of applications are available for use with your device. Either way, I want to spend a moment pointing out a few J2ME applications that are available as of this writing.

J2ME applications can be grouped into the following three types:

- Server applications
- Application portals
- End-user applications

This book has focused almost entirely on end-user J2ME applications, which are basically just MIDlets. You can also think of an end-user J2ME application as a client application because it runs directly on a Java-powered device. The next few sections examine a few J2ME application offerings of each of these types.

Server Applications

J2ME server applications are designed to run on a Web server, and typically facilitate the sale and distribution of end-user J2ME applications. Server applications aren't necessarily written in Java; their significance is that they improve the process of making MIDlets readily available to end users. J2ME server applications are primarily involved in the delivery of J2ME end-user applications; therefore they are typically used by wireless providers as opposed to individuals.

An interesting J2ME server application is AppStream, which supports a new concept known as *application streaming*. Application streaming is quite interesting because it allows you to begin using an application while it is still being downloaded. You can think of application streaming as the application equivalent of media streaming, which is very popular for delivering streamed audio and video content over the Internet. AppStream's server application makes it possible to deliver MIDlets to end users in a very efficient manner that minimizes the effect of slow wireless network connections. To learn more about Appstream's wireless application streaming, visit the AppStream Web site at `http://www.appstream.com/`.

NEW TERM *application streaming*—The processing of delivering an application so that a user can begin using the application while it is still being downloaded.

Another J2ME server application is the 4thpass Mobile Application Server (MAS), which provides a means for a wireless service provider to deliver wireless applications to endusers. One unique feature of the 4thpass server application is its on-the-fly size reduction capabilities, which are designed to decrease the download time required to obtain a MIDlet over a wireless connection. To find out more about the Mobile Application Server, visit the 4thpass Web site at `http://www.4thpass.com/`.

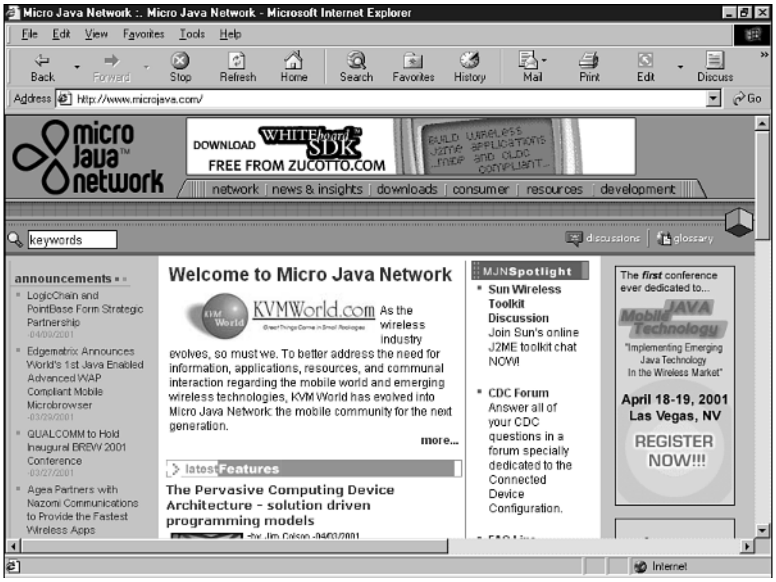
Application Portals

Whereas J2ME server applications provide the server-side infrastructure for delivering MIDlets to device users, there are also J2ME portals, which are Web sites devoted to the distribution of end-user J2ME applications. These sites serve as both freeware and shareware MIDlet repositories, as well as places to purchase and download other commercial MIDlets. Although device users will probably initially rely on the application distribution model made available by their device vendor or wireless service provider, eventually J2ME application portals will likely become popular locations from which to add new applications to J2ME devices.

NEW TERM *J2ME portal*—A Web site devoted to the distribution of end-user J2ME applications.

One particularly neat J2ME portal is the Micro Java Network, located at `http://www.microjava.com/`. In addition to providing industry news and message forums, the Micro Java Network has several libraries of J2ME applications that are available for download. There is also a good developer section of the site that you might want to visit at some point. Figure 20.5 shows the Micro Java Network home page.

FIGURE 20.5
The Micro Java Network Web site serves as a J2ME portal for accessing news and forums and downloading J2ME applications.


Micro Java Network :. Micro Java Network - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.microjava.com/
Go
micro Java network
DOWNLOAD WHITEboard SDK FREE FROM ZUCOTTO.COM
BUILD WIRELESS J2ME APPLICATIONS MIDP AND CLDC COMPLIANT
network | news & insights | downloads | consumer | resources | development
keywords
discussions | glossary
Internet

announcements

- LogicChain and PointBase Form Strategic Partnership -04/09/2001
- Edgematrix Announces World's 1st Java Enabled Advanced WAP Compliant Mobile Microbrowser -03/29/2001
- QUALCOMM to Hold Inaugural BREW 2001 Conference -03/27/2001
- Agea Partners with Nazomi Communications to Provide the Fastest Wireless Apps

Welcome to Micro Java Network

KVMWorld.com Great Things Come in Small Packages As the wireless industry evolves, so must we. To better address the need for information, applications, resources, and communal interaction regarding the mobile world and emerging wireless technologies, KVM World has evolved into Micro Java Network: the mobile community for the next generation.

more...

latest Features

The Pervasive Computing Device Architecture - solution driven programming models

-by Jim Colson -04/03/2001

MJNSpotlight

- **Sun Wireless Toolkit Discussion** Join Sun's online J2ME toolkit chat NOW!
- **CDC Forum** Answer all of your CDC questions in a forum specially dedicated to the Connected Device Configuration.

The *first* conference ever dedicated to... JAVA Mobile Technology "Implementing Emerging Java Technology In the Wireless Market" April 18-19, 2001 Las Vegas, NV REGISTER NOW!!!

Although the Micro Java Network is primarily focused on providing freeware and shareware MIDlets for download, another portal is geared toward making MIDlets available for purchase. This is MIDlet Central, which is touted as "The World's First Wireless Java Café." The MIDlet Central Web site is located at `http://www.midletcentral.com/`, and is made possible by Digital Mobility. The eventual goal of MIDlet Central is to enable independent software vendors to upload MIDlets that can then be purchased by endusers. As of this writing, the site is relatively new and doesn't yet have any MIDlets available for purchase. However, a few MIDlets are available for free download. Figure 20.6 shows the MIDlet Central home page.

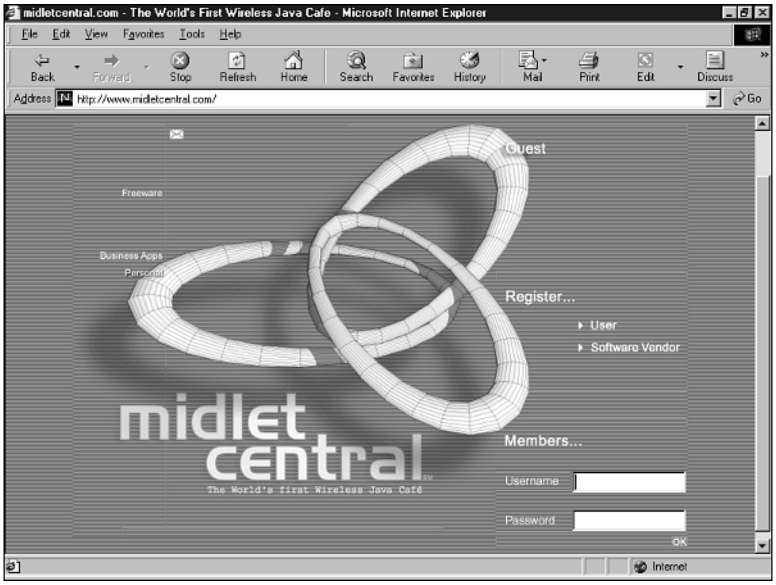

midletcentral.com - The World's First Wireless Java Cafe - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://www.midletcentral.com/
Go
Freeware
Business Apps
Personal
Guest
Register...
User
Software Vendor
Members...
Username
Password
OK
midlet central
The World's first Wireless Java Café
Internet

FIGURE 20.6
The MIDlet Central Web site serves as a J2ME portal for purchasing J2ME applications.

End-User Applications

The last type of J2ME application is perhaps the most important to you, the J2ME developer. This is end-user applications, which in the case of MIDP devices are MIDlets. You are actually quite familiar with J2ME end-user applications because you have spent the previous nineteen lessons developing them. Even so, it's worth seeing what else is out there in terms of commercial MIDlets.

AGEA's MobileDASH

Perhaps the most significant commercial MIDlet offering as of this writing is MobileDASH, which is a MIDlet application suite made available by AGEA Corporation (formerly agentGo). MobileDASH is a suite of MIDlets that address many of the common productivity tasks associated with a mobile device. More specifically, the MobileDASH product is broken down into the following modules:

- **My Contacts**—Contact manager that enables the user to call, alert, or e-mail people in his contact list
- **My Schedule**—Personal scheduler that keeps track of scheduled events
- **My Files**—File manager that provides access to a server file system
- **My Team**—Instant messenger that enables the user to send real-time messages to other people
- **My Alerts**—Notification service that delivers personalized alerts to the user
- **My Expenses**—Expense manager that enables the user to enter expense information that can later be integrated with expense report software
- **My Bookmarks**—Favorites list that contains the user's favorite Web sites

As you can see, the MobileDASH application addresses most of the productivity needs of mobile device users who are on the go. It isn't clear how application suites such as MobileDASH will fare against similar offerings for handheld computers such as Palm devices. Palm users are already accustomed to storing this kind of information on their Palm devices, and it isn't clear if they will be willing to use their phone or pager instead. On the other hand, MobileDASH is supported on the Palm as well, so the option exists to use the application suite on a phone, pager, or Palm device, or possibly some combination of the three.

To find out more about the MobileDash application suite, visit the AGEA Web site at `http://www.agea.com/`.

4thpass KBrowser

Earlier in the lesson I mentioned 4thpass as the software vendor for a J2ME application server. 4thpass also offers an end-user application that is quite interesting. This application is called KBrowser, and is essentially a Web browser designed for use on mobile devices. The term used to describe KBrowser is *micro-browser*, which refers to the fact that the browser is designed to execute within a very limited hardware environment. More specifically, micro-browsers are designed to accommodate the limited memory, processing power, and screen sizes of mobile devices. Most micro-browsers use the Wireless Application Protocol (WAP), which enables them to display Web pages developed in WML (Wireless Markup Language), a simplified version of HTML.

NEW TERM *micro-browser*—A Web browser designed to operate within the constrained environment associated with mobile devices.

Although the KBrowser J2ME application is a micro-browser, it packs a fair amount of features into its small software footprint. For one, KBrowser supports the download and installation of other J2ME applications through its own interface. If your mobile device doesn't already include a wireless Web browser of some sort then KBrowser is something you might want to look into. To learn more about KBrowser, visit the KBrowser Web site at `http://www.4thpass.com/kbrowser/`.

Summary

Even the most powerful and compelling of MIDlets wouldn't be of much use if there weren't real devices on which to run them. Granted, it's fun to play with the J2ME emulator and use it as a test bed during MIDlet development, but at some point MIDlets must be set free to roam the wireless world on physical devices. This lesson explored the different devices that offer support for J2ME and MIDlets right now, as well as a few devices that are on the horizon. By learning about these devices, hopefully you grasped the potential of how many users out there might want to use your MIDlets.

In addition to assessing the marketplace for J2ME devices, this lesson also explored the current crop of J2ME applications. Admittedly, things are still somewhat thin at the moment, but many companies and individuals alike are gearing up to offer J2ME applications in the very near future. The next lesson peeks into a Java-powered crystal ball and attempts to unravel the mysteries that lie ahead for J2ME.

Q&A

Q Are the three device types mentioned in this lesson the only J2ME devices available?

A No. This book is focused on the CLDC and MIDP APIs, which are geared toward wireless mobile devices. The three device types mentioned in this lesson (mobile phones, pagers, and handheld computers) are the three main types of wireless mobile devices, but there are other devices that can certainly be supported by J2ME. In fact, the next lesson introduces you to the Connected Device Configuration (CDC), which targets an entirely different class of devices from the CLDC. Even within the CLDC, it is possible that entirely new devices will emerge that won't necessarily fit into the three types I laid out in this lesson.

Q How do I know if my mobile phone will enable me to run MIDlets?

A First, check and see if the phone is specified as supporting Java. Because mobile phones collectively fall under the MIDP device classification, it is reasonable to expect that all Java-powered phones are capable of running MIDlets. Phone vendors might not use the terminology of "J2ME," and will probably instead just say

"Java" when referring to their phone's Java features. However, if Java is involved with a mobile phone, it is safe to assume that it is J2ME with the MID Profile.

Q How do I begin developing MIDlets specifically for Palm devices?

A Because MIDlets are designed to run across all MIDP environments, all you must do is develop your MIDlets according to the rules already set forth in this book. In other words, stick to the CLDC and MIDP APIs and you should be OK. To run MIDlets on your Palm device, you'll need to obtain a Palm-specific KVM.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What is the difference between a J2ME server application and a J2ME end-user application?
2. What is the significance of application streaming?
3. What is a J2ME portal?

Exercises

1. Visit some of the Web sites mentioned throughout this lesson to become acquainted with the different J2ME devices and applications mentioned.
2. If you have any MIDlets of your own that you're considering developing, think about how you might like to distribute them. Freeware? Shareware? Or maybe you want to distribute them commercially using a MIDlet portal.

DAY 21

The Future of J2ME

Just as Marty McFly in the 1985 hit movie *Back to the Future* was able to travel to the future and see a glimpse of what life was to be like, I want to spend this last day giving you a glimpse of the future of J2ME. Unfortunately, I don't have Dr. Emmett Brown or a stainless steel DeLorean sports car to help in my efforts. Even so, I think you'll find this day to be an interesting change of pace because it attempts to paint a picture of what the future has in store for J2ME and wireless Java devices.

The day begins by comparing the CLDC to the CDC (Connected Device Configuration), which ups the ante on the CLDC to accommodate devices with a little more power. You then learn how J2ME relates to several other emerging Java technologies such as Jini and JavaCard. The rest of the day addresses a wide range of J2ME topics such as its impact on gadgets, gaming, and multimedia. The following major topics are covered in this lesson:

- Comparing the CLDC to the CDC
- Assessing the relationship between J2ME and other emerging Java technologies
- Exploring the role of J2ME in futuristic gadgets

- Understanding how J2ME is factoring into the world of video games
- Experiencing the Multimedia Capabilities of J2ME

The CLDC Is Not Alone

Throughout this book thus far you have focused on the development of special Java programs called MIDlets that are designed to run on devices that adhere to a specific set of APIs. These APIs are clearly identified in the CLDC (Connected Limited Device Configuration) and MIDP specifications. Together, these specifications describe a configuration and a profile for a class of mobile wireless devices that includes mobile phones, pagers, and some handheld computers. Although the CLDC is the first configuration created for J2ME, it is not the only one.

In addition to the CLDC, J2ME also has the CDC, which is the Connected Device Configuration. Notice that the word "Limited" is missing from the name of this configuration. This should tell you something about the kinds of devices that it targets. The CDC also applies to connected devices, but CDC devices are expected to include significantly more memory and processing power than CLDC devices. More specifically, CDC devices are required to have the following:

- 512Kb of non-volatile memory
- 256Kb of volatile memory
- No power limitations

The obvious difference between CDC and CLDC devices is the increased memory requirements of the CDC. If you recall from Day 1, "Java 2 Micro Edition: The Big Picture," the CLDC only requires devices to have 160Kb of total memory. As the preceding list shows, the total memory (non-volatile and volatile) of a CDC device must be at least 768Kb. The other difference between the two types of devices revealed in this list is the assumption of a steady power source in CDC devices. This is a significant difference that has to do with the fact that practically all CLDC devices are battery powered, and therefore are very capable of intermittently losing power. Most CDC devices will likely be plugged into a more steady power source.

Alluding to the steady power source in CDC devices, Sun has described them in an interesting way as "devices that you plug into the wall." Although other types of CDC devices will certainly emerge, this description hints that set-top Internet boxes are one of the main types of devices that fall under the CDC. Many smart appliances will also likely be classified as CDC devices. However, a few devices don't necessarily plug into a wall. The most obvious are the high-end handheld computers such as Pocket PCs that clearly have more memory, more processing power, and larger screens than the typical CLDC device, so it makes sense for them to include CDC support.

Another big difference between the CLDC and the CDC is that the CLDC relies on the K virtual machine (KVM), whereas the CDC requires a full-blown Java virtual machine, which is now being referred to as the *C virtual machine (CVM)*. If you recall, the KVM is a scaled-down virtual machine that is designed to run more efficiently within the constraints of compact devices with limited memory and processing power. The full Java virtual machine required by the CDC is the same virtual machine that is used in J2SE and J2EE environments. Clearly, the CDC describes a more complete Java runtime environment than the CLDC, which isn't too surprising.

NEW TERM *C virtual machine (CVM)*—A full-featured Java virtual machine that supports all of the functionality of the Java language.

I point out the differences between the CLDC and the CDC because you will likely begin to see new profiles that target the CDC and that have significantly more capabilities than the Mobile Information Device Profile (MIDP). The MIDP profile is based on the CLDC, and is suitably constrained for devices with limited capabilities. As J2ME continues to evolve and grow in popularity, the other profiles will appear in addition to the MID profile. At some point you might have to decide whether it's worth the extra development effort to modify a MIDlet so that it is capable of running under a different profile. Many API features specified in the MID profile may be supported in other profiles, but there are no guarantees.

Note

The first profile announced for the CDC is the Foundation profile, which targets Web-enabled consumer devices such as set-top Internet boxes.

The Convergence of J2ME and Other Java Technologies

As you might know, the software architects at Sun have been quite busy since the inception of Java several years ago. Numerous Java-related technologies have rolled out of Sun, some of which have caught on and prospered, and others that have fizzled. Obviously, J2ME is one of those technologies that appears to be here to stay. However, some other relatively new Java technologies seem to be related to J2ME. It's important to understand how these technologies compare to J2ME, and whether they will be involved in the future of J2ME.

The Demise of PersonalJava

Many months ago, before the birth of J2ME, Sun unveiled a technology called *PersonalJava* that was aimed at applying Java to consumer devices such as phones and appliances. Sound familiar? PersonalJava was no doubt the precursor to J2ME—it even includes its own API that is geared toward devices and appliances, as opposed to full-blown standalone computers. Logic would tell you that J2ME was released as a replacement for PersonalJava, but as of this writing PersonalJava is still alive and well.

A closer look at PersonalJava reveals that it does in fact target some of the same devices targeted by J2ME. The primary difference is that J2ME is more granular because of its configurations and profiles. However, these devices don't fall under the CLDC or MIDP specifications because they are more powerful than CLDC and MIDP devices. Therefore, PersonalJava targets the devices that don't quite fit into the MID profile, which means that they require a profile of their own to fit within J2ME. Not to let a Java need go unfulfilled, Sun recently announced the Foundation profile under the CDC, which is designed to replace PersonalJava.

New Term *Foundation profile*—A CDC profile that targets networked consumer devices and will ultimately replace the PersonalJava technology.

The bottom line is that PersonalJava is going away, but it is being replaced by the Foundation profile in the CDC. This move basically brings PersonalJava within the realm of J2ME, which is both a logical and a necessary move to organize the "micro" Java technologies. If you've done any research or work with PersonalJava, you might want to look into the Foundation profile. For more information on this profile, visit the Foundation profile page on Sun's Java Web site at `http://java.sun.com/products/foundation/`.

Networking with Jini

If you have any experience with distributed network systems, you probably know that Jini (pronounced "Genie") isn't just a new way to spell the name of a guy that lives in a bottle. In fact, *Jini* is a Java technology that is aimed at integrating software services across a distributed network. More specifically, Jini makes it possible to integrate and maintain network components such as applications, servers, and databases that are being run on a wide range of different networked computers and devices. As this description reveals, Jini is quite broad in scope, and will likely impact networked systems on a large scale.

New Term *Jini*—A Java technology that provides a means of integrating and managing distributed network services.

Currently there is no direct relationship between Jini and J2ME. Existing implementations of Jini are designed to run within J2SE or J2EE environments, so a J2ME implementation doesn't currently exist. However, there is no reason why Jini couldn't be applied to J2ME systems. In fact, one of the main marketing angles of Jini was to help solve many networking problems associated with embedded systems. Compact devices are typically involved in these kinds of systems, so it only makes sense to eventually provide an implementation of Jini for J2ME.

Java Card and J2ME Smart Cards

Perhaps one of the most interesting new Java technologies is Java Card, which is an API designed to provide a means of creating applications for smart cards. A *smart card* is a small plastic card the size of a credit card that houses a tiny computer capable of running compact applications. Smart cards are being designed to perform a variety of functions, the most common of which is to act as electronic cash, meaning that you can use a smart card to pay without the transaction relying on a bank account. Money is electronically transferred onto the card from a bank account, where it can then be spent just as if it were physical cash.

New Term *smart card*—A small plastic card the size of a credit card that houses a tiny computer capable of running compact applications.

Note One of the first Java-powered smart cards to hit the market is the Blue card from American Express, which utilizes PersonalJava.

Java Card is a Java API through which it possible to develop smart card applications in Java. Of course, a Java runtime environment must exist on the smart card in order for Java Card applications to run. Care to guess what kind of runtime environment this might entail? Well, up until now J2ME has not entered the Java Card picture, but it makes sense that it could at some point in the future. Java Card currently uses its own virtual machine, known as the Java Card virtual machine, or JCVM. However, it isn't out of the question to imagine that the KVM could be applied to Java Card. If the KVM somehow won't work, maybe Java Card itself will eventually appear as a new profile under the CLDC. Sun hasn't made any formal declarations regarding any future merger of Java Card and J2ME, but it seems like a logical fit to me. To learn more about Java Card, visit the Java Card section of the Java Web site at `http://java.sun.com/products/javacard/`.

New Term *Java Card*—A Java technology that makes it possible to develop smart card applications in Java.

J2ME: A Gadget Lover's Dream

J2ME opens up the possibilities to apply Java to a wide range of interesting devices, so it shouldn't come as too much of a surprise that some very intriguing Java-powered devices are slated for release in the very near future. These devices range from cooking appliances to gas pumps, and all stand to have a direct impact on our lives in ways that don't typically involve traditional computers. In other words, J2ME is finally fulfilling Sun's original vision of Java as a technology to add intelligence and interactivity to everyday devices.

Java In the Smart Home

Smart homes have been around for a while, and a variety of different technologies are available for automating tasks within the home and establishing a network of appliances. Thanks to J2ME, Java is now entering the smart home picture as a means of creating custom applications that can be executed within appliances and other devices in a smart home network. A single server that manages all of the major "smart" systems in the home of the future will likely drive the Java-powered home. Different appliances and devices can then utilize J2ME to communicate with the server and provide a high degree of automation and interactivity.

I have not seen any production Java-powered appliances on the market just yet, but several interesting prototypes have appeared that present some unique possibilities for a Java smart home. An oven is capable of receiving cooking instructions directly from a set-top Internet box. This means that you'll no longer need to scramble for a piece of paper to write down a recipe when you're watching your favorite cooking show. The concept of appliances communicating with each other, as in this example, is at the heart of Java and its application to smart homes. Exactly how J2ME will fit into the smart home equation, or even if Java can become a dominant technology for appliances, is not clear. However, the quick adoption of J2ME in other products such as wireless mobile devices will definitely help cement it as a viable option for smart appliances.

Pumping Gas with J2ME

The next time you stop to fill up your car with gasoline, consider the fact that J2ME might be lurking behind the credit card swiper. Actually, only a few trial Java-powered gas pumps are in operation as of this writing, but they are being considered as a means of enabling gas pumps to communicate better with accounting systems, and also provide information such as driving directions and entertainment suggestions. Of course, because you're captive for a couple of minutes pumping gas this is an ideal time to bombard you with advertising. So, a significant goal of the Java-powered gas pump is to provide a means of presenting advertisements to customers.

A more ambitious possibility for Java gas pumps is to enable the pump to communicate with your automobile, in which case it can perform electronic diagnostics and notify you of service issues. This would require automotive manufacturers to include support for such a system, which is something that isn't likely to happen in the very near future. On the other hand, most luxury automobiles now offer computer navigation systems, which could probably be modified to include support for interfacing with an intelligent gas pump.

One other interesting facet of Java-powered gas pumps is that they also will likely involve the Java Card technology. Keep in mind that by using certain kinds of Java Cards you could carry out financial transactions, and these Java Cards could therefore be useful in purchasing gas. Beyond that, I wouldn't rule out the possibility of Java Cards used primarily as information storage units. For example, a Java Card could be used to store your car's service history—each time you visit a gas pump you could swipe the card to find out and possibly even schedule your next service visit.

Note

Automakers GM, Ford, BMW, and Fiat have all developed concept automobiles that utilize Java technology. It's now a matter of waiting to see who is first to include Java in a production vehicle.

Java on Mars?

No doubt the most surprising applications of Java, at least to me, is the inclusion of Java technology into Mars exploration. NASA created an applet that enables scientists to collaborate and coordinate the movements of a simulated Mars rover. This is only a simulation, so Java isn't currently being used to drive a rover around Mars. Also, the Java technology to which I refer is actually a Java applet, which definitely doesn't involve J2ME. However, if Java can be used to simulate the control of a Mars rover, I don't think it's out of the question to propose that J2ME might somehow fit into the control system of a real Mars rover.

Granted, I could be daydreaming a bit by suggesting that NASA might realistically consider using J2ME as part of the control software for a Mars rover. However, when you consider the requirements of practically any space-based vehicle or device, J2ME makes a lot of sense. Any kind of device that you must transport over a distance that great must be extremely compact, and therefore have limited capabilities. This is exactly the type of device for which J2ME is intended. Are you reading this, NASA?

Gaming with J2ME

In the previous lesson I mentioned that the Java-powered Motorola phones feature a puzzle game created by Sega. As a sure sign of the video game industry's realization that J2ME is an important technology, Sega has entered into an agreement with Motorola to develop several games using J2ME that will run on the Motorola phones. It is pretty clear that an enormous market exists for relatively simple games that people can play on their mobile devices. How often do you find yourself in a situation where you are waiting for a few minutes and need a quick distraction? A quick game of *Frogger* could be just what you need to blow off a little steam and clear your head.

In addition to the large market for mobile games, Sega also views J2ME games as incredibly simple to produce, at least in comparison to computer and console video games. The production of a modern video game requires teams of artists, programmers, and writers, not to mention an enormous amount of man hours and other resources. The simplicity of mobile games is very appealing to game publishers because it enables them to turn out games quickly and with much less overhead.

In terms of specific games that you might want to be on the lookout for, Sega has announced the following games as part of their agreement with Motorola, in addition to the Borkov puzzle game:

- Golf
- Blackjack
- Columns (similar to Tetris)
- Sonic Logic
- Sonic Head-On
- Sonic's Bomb Squad

Clearly, Sega is relying on at least one of their successful brands in Sonic the Hedgehog. This is another advantage for current game makers because mobile gaming makes it possible to leverage existing games that are popular on other platforms. It isn't clear at the moment if other game publishers are taking notice of the Sega/Motorola partnership, but I wouldn't be surprised to see other big players in the video game industry embrace J2ME in the near future and begin developing mobile games.

J2ME and Multimedia

You probably wouldn't expect me to mention multimedia as an area where J2ME is making inroads. The fact is that most multimedia content (audio, video, and so forth) takes up a lot of space, and therefore would prove difficult to fit into the limited memory constraints of many J2ME devices. However, a few multimedia technologies are taking advantage of J2ME to offer unique services to users of mobile devices. The following sections highlight some examples of these technologies.

Wireless Video

Motorola has their fingers into many mobile technologies, and this is not surprising because they are in the business of manufacturing and selling mobile devices. Knowing this, it might come as a little surprise to find out that Motorola is an investor in a company called PacketVideo, which is involved in bringing streaming video content to wireless mobile devices. PacketVideo has developed an implementation of the MPEG-4 video standard that makes it possible to send streaming video content over a wireless network connection to mobile devices. The mobile PacketVideo software is based upon J2ME, which makes it possible to view streaming video such as news clips and movie trailers on a wireless mobile device. For more information about PacketVideo, check out the PacketVideo Web site at `http://www.packetvideo.com/`.

Musical Phones

Even if you aren't much into music, I'm sure you've heard about the manner in which MP3 music files have turned the music industry on its ear. Just as the JPEG image format makes it possible to efficiently store photographs digitally on a computer, the MP3 music format makes it possible to store music. Although the battle between shared music advocates and the music industry has been going on for some time now, it's unlikely that we'll see any real resolution in the near future. The reality is that digital music represents a painful shift for an industry that hasn't had to change much over the years. Eventually we are likely to see a system that permits people to purchase and download individual songs for a nominal fee. Granted, such systems already exist, but not on a widely accepted scale.

Believe it or not, digital MP3 music is factoring into J2ME, thanks to a company called Sensate, the creators of an application called the AoIP Wireless Media Server. This application functions as a wireless media server that permits users to download MP3 music to portable devices such as MP3 players, handheld computers, car stereos, and even phones and pagers. Although the AoIP server does a lot of the work involved in delivering wireless digital music, it requires each mobile device to have a client with which it can communicate. And that's where J2ME enters the picture. To learn more about Sensate and the AoIP Wireless Media Server, visit the Sensate Web site at `http://www.sensateinc.com/`.

Note

> Although I focused on music as the type of digital audio delivered using the AoIP Wireless Media Server, it is actually capable of delivering any digital audio stored in the MP3 format. This could also include personal memos that you record yourself, as well as audio books that you purchase.

Summary

Although you didn't hack through any Java code today, you hopefully gained some perspective on the direction that J2ME is headed. It's a suitable way to end a book such as this by focusing on the future of the technology and attempting to point out some of the interesting ways in which J2ME is being applied to different industries. When you think about it, we live in a world full of devices. Currently, not many of these devices are equipped to perform more than simple, mundane processing, and even fewer of them are capable of communicating with one another. J2ME is attempting to empower these devices with more brains and better social skills, and hopefully improve our lives along the way.

This lesson began exploring the future of J2ME by introducing you to the CDC (Connected Device Configuration), and clarifying its relationship to the CLDC. You learned about the impact J2ME is having on other Java technologies such as Personal Java, Jini, and Java Card. You then explored some of the ways in which Java is being introduced into various gadgets such as smart appliances and Java-powered gas pumps. From there, you found out how the video game industry has suddenly shown an interest in J2ME. Finally, the lesson concluded by assessing the role of J2ME in a couple of mobile multimedia technologies.

Q&A

Q Is it safe to assume that J2ME will stay more or less the same in the future?

A Yes and no. J2ME is a stable technology that is defined in general enough terms that it is unlikely to change dramatically at its core. However, I fully expect to see additional configurations and profiles added to J2ME as new devices are introduced with different needs than are currently addressed by J2ME.

Q Where can I go to find out the latest news related to J2ME?

A The best thing to do is go straight to the source, the J2ME Web site at Sun. This Web site is located at `http://java.sun.com/j2me/`. There you'll find plenty of useful information, including late breaking press releases on happenings in the J2ME world.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice. The answers to the quiz are in Appendix A, "Quiz Answers."

Quiz

1. What is the difference between the CLDC and the CDC?
2. What is the C virtual machine (CVM)?
3. What is a smart card?

Exercises

1. Look around your home and assess the different devices that you use on a daily basis. Consider how many of them could benefit from J2ME by incorporating automation features and the capability to communicate with each other.
2. Think of a unique application of J2ME that I haven't mentioned anywhere in the book. Now work through the general design of this application, taking into account the capabilities of J2ME. Is this something you'd like to pursue as a project of your own, or maybe as part of your work?

Part VI

Appendixes

A Quiz Answers

B What's on the CD-ROM

APPENDIX A

Quiz Answers

Quiz Answers for Day 1

1. The three editions of Java that are currently available are J2SE, J2EE, and J2ME.
2. A configuration is a minimum set of APIs that is useful for developing applications to run on a range of devices. A standard configuration for wireless devices is known as the Connected Limited Device Configuration, or CLDC.
3. A profile is a specific set of APIs that target a particular type of device. A profile sits on top of a configuration and provides more detail; a configuration describes in general terms a family of devices, while a profile gets more specific and isolates a particular type of device within that family.
4. The acronym MIDP stands for Mobile Information Device Profile.

Quiz Answers for Day 2

1. A compiled MIDlet class must be verified using a verification tool before it can be executed in a wireless environment.
2. The virtual machine in mobile devices is called the K Virtual Machine, or KVM for short.

3. The two primary development tools included in the J2ME Wireless Toolkit are the bytecode verifier and J2ME emulator.
4. The purpose of the configuration editor included in the Motorola SDK for J2ME is to allow the editing and creation of custom device profiles.

Quiz Answers for Day 3

1. Two parameters of a MIDlet's execution that cannot be accounted for in the J2ME emulator are execution speed and available memory.
2. The J2ME device profile that maps the letter keys on the keyboard to device keys is the pager device profile.
3. The J2ME emulator's `kvem.home` system property is used to identify the root directory of the J2ME Wireless Toolkit.
4. The pieces of MIDlet information that can be traced by turning on certain emulator properties include garbage collection, method calls, class loads, and exceptions.

Quiz Answers for Day 4

1. The purpose of the MIDlet pre-verification process is to take some of the load off the virtual machine when it loads a MIDlet and performs a security check on the MIDlet classes.
2. The three lifecycle methods associated with the `MIDlet` class are `startApp()`, `pauseApp()`, and `destroyApp()`.
3. There is exactly one `Display` object for each MIDlet that is executing on a device.
4. The significance of a MIDlet command's priority is that it determines the placement of the command for user access.

Quiz Answers for Day 5

1. None of the classes in the standard J2SE `java.awt` package are inherited in the CLDC/MIDP APIs. There is no AWT in MIDlet programming; instead, the `javax.microedition.lcdui` package provides similar support for constructing GUIs.
2. The Generic Connection Framework (GCF) consists of a set of connection interfaces, along with a `Connector` class that is used to establish the different connections.

3. The `MIDlet` class is located in the `javax.microedition.midlet` package, and is the only class in the package.
4. A record store is a simplified database for MIDlet data storage, and support for record stores is found in the `javax.microedition.rms` package.

Quiz Answers for Day 6

1. A device profile is a software description of a physical MIDP device.
2. The Configuration Editor tool generates a text file containing a list of device properties. This file has a `.props` file extension and is known as a properties file.
3. It is necessary to define the dimensions of a device's display area as part of the device profile because the J2ME emulator uses this area to render the output of MIDlets that are executed using the device profile. In other words, the size of the display area dictates the size of the emulation screen.
4. The three files that are necessary to use a custom device profile within the Motorola SDK for J2ME are the properties file for the device profile, the device image file, and an execution script for the device profile.

Quiz Answers for Day 7

1. The MIDP graphics coordinate system has its origin located in the upper-left corner of the device screen.
2. Okay, I admit this was a trick question. Unlike standard Java graphics programming, there is no `Color` object in the MIDP API. Instead, you specify colors as three integers representing the primary colors (red, green, and blue) or by a single integer containing a color in the hexadecimal form `0x00RRGGBB`.
3. To draw a perfect circle using the `Graphics` class, you must call the `drawArc()` method, specify an equivalent width and height, and set the sweep angle of the arc to 360 degrees.
4. To draw graphics in a MIDlet, you must derive a class from `Canvas` that implements the `paint()` method and then set an instance of that class as the current display for the MIDlet.

Quiz Answers for Day 8

1. Every MIDlet has a unique instance of the `Display` class, which represents the device display and user input controls for the MIDlet.

2. The only two direct subclasses of the `Displayable` class are `Canvas` and `Screen`.
3. For a GUI component to be capable of being placed on a form, it must be derived from the `Item` class.
4. The primary difference between the `List` class and the `ChoiceGroup` class is that `List` is a screen and `ChoiceGroup` is an item. This means that a `ChoiceGroup` object can be added to a form, but a `List` object cannot.

Quiz Answers for Day 9

1. The Connector class is the fundamental GFC networking class used to establish all MIDlet network connections.
2. The only type of network connection guaranteed to be supported by all MIDP implementations is an HTTP connection.
3. To establish an HTTP connection, you must pass the URL of the Web resource into the `Connector.open()` method.
4. To obtain an input stream on an established connection, you must call the `openInputStream()` method on the connection and cast the resulting object to the appropriate stream type.

Quiz Answers for Day 10

1. The Weather.gov Web site is particularly well suited for use with the Weather MIDlet because the weather data it offers is simply formatted and easy to process programmatically.
2. It is necessary to convert the two-letter state abbreviation before passing it into the `getConditions()` method because the abbreviation is used to form a directory name within the weather Web page URL, which must be specified in lowercase.
3. The `done` flag serves two purposes. It provides a shortcut out of the main `while` loop after the line of the Web page containing the weather information has been found. It also provides a means of detecting whether an error has occurred while obtaining the weather data.
4. If for some reason the weather information cannot be found for the city and state entered in the Weather MIDlet, an error message is displayed.

Quiz Answers for Day 11

1. The MapQuest driving directions are more suitable for use with a MIDlet than the MapBlast directions because they are presented as raw, unformatted text. In other words, there are no HTML tags (`<b>`, `</b>`, and so forth) embedded directly in the text for each step of the directions.
2. The Directions MIDlet finds the text for each step of the driving directions within the HTML code for the Web page by looking for a recurring pattern of text. This text pattern is actually the number of the step, which appears in bold, and is therefore marked up with HTML code using the `<b>` and `</b>` tags. When you encounter this line of code, you know the step text is always on the next line.
3. No components are added to the directions screen when the rest of the MIDlet's user interface is initialized because the directions screen operates as a dynamic form. This means that components (string items) are only added to the directions screen as needed when the direction steps are read from the driving directions Web page. Likewise, these same components are removed from the directions screen when the user returns to the locations screen.
4. The purpose of the `replaceSpaces()` method is to replace space characters in a string with special placeholder characters that are acceptable to use in a URL.

Quiz Answers for Day 12

1. The three major areas of code optimization are maintainability, size, and speed.
2. The least important type of optimization for MIDlets is maintainability optimization because it stresses the understanding and organization of code in a MIDlet over the size or performance of the code.
3. It is possible to avoid using objects in a MIDlet by using primitive data types (for example, `int`, `boolean`, and `char`) instead.
4. Object recycling is the process of reusing an existing object instead of creating a new one, which avoids an unnecessary memory allocation.

Quiz Answers for Day 13

1. The fundamental unit of storage in the RMS is a record.
2. Record IDs serve as the primary key for record stores, and are used to uniquely identify records.

3. The `javax.microedition.rms` package is the MIDP package that houses the classes and interfaces for the RMS API.
4. A record is stored in a record store as an array of bytes.

Quiz Answers for Day 14

1. The contact information for a contact record is stored in the contact record store as an array of bytes. However, a string representation of the contact information is used as the basis for this array of bytes, and the string record includes the pieces of contact information separated from each other by semicolons.
2. The image icon for a contact is associated with the contact in the main contact list by passing the icon image as the second parameter of the `append()` method when adding the contact to the main `List` component.
3. You keep the user from selecting the `Delete` command while adding a contact by removing the command from the contact screen when a contact is being added, and then adding it back if a contact is being edited.

Quiz Answers for Day 15

1. The main benefit to using a mobile electronic check register is that you can enter transactions at the time when you make them. Additionally, an electronic check register automatically calculates a running balance, which eliminates human error and provides you with a current account balance at all times.
2. The CheckRegister MIDlet shows the balance to the user as the last item in the main transaction list. Because the balance is in the same list with the transactions, it is important in the MIDlet code to make sure that the user cannot interact with the balance item as if it were a transaction.
3. It is necessary to store the transaction amount in cents because J2ME doesn't support floating-point operations. In other words, it isn't possible to work directly with numbers that have a decimal place. So, the workaround is to always deal with transaction amounts in cents so that there never is a decimal place to deal with. Of course, this requires some care when it comes time to display an amount to the user or accept an amount entered by the user, but it still is fairly straightforward in the code.

Quiz Answers for Day 16

1. Unlike most traditional offline auctions, most online auctions last several days and are carried out entirely through a Web interface.
2. The purpose of an item ID is to uniquely identify an item in an online auction.
3. The two pieces of item information that are stored away persistently for the AuctionWatch MIDlet are the item ID and description.

Quiz Answers for Day 17

1. The two main types of animation are frame-based animation and cast-based animation (sprite animation). Frame-based animation simulates movement by displaying a sequence of pregenerated, static frame images, while cast-based animation simulates movement using graphical objects that move independently of a background.
2. A transparent color is a color in an image that isn't drawn when the rest of the colors in the image are drawn.
3. Flicker is a negative visual effect associated with animation that occurs when a portion of the screen is erased and redrawn rapidly.
4. Double buffering is a technique of rendering animation frames that involves erasing and drawing to an offscreen image, and then drawing the complete results to the screen at once, thereby eliminating flicker.

Quiz Answers for Day 18

1. Traveling Gecko is based on the classic arcade game Frogger. Not only is Frogger a classic in the minds of those of us who grew up feeding quarters to arcade games, it was also later immortalized in an episode of the hit television show "Seinfeld."
2. Each time the player safely guides the gecko across the desert, he is rewarded with 25 points.
3. A real gecko in the same situation as our virtual traveling gecko would fare quite well. Geckos are extremely fast—considerably faster than your Java-handicapped gecko.
4. The rocks are modeled as sprites because they must serve as a barrier for the gecko, which requires detecting collisions with the gecko. Logically, you could think of the rocks as part of the background, in that they don't do much beyond limiting the gecko's movement. However, if the rocks were just drawn on the back-

ground, there would be no straightforward way to detect collisions between them and the gecko, which means you couldn't limit the gecko's movement.

Quiz Answers for Day 19

1. Weighting is a method of looking at a game at any point and calculating a score for each player.
2. The map holds the lookup table of winning combinations, which is necessary to calculate the score for each player at any given point in the game.
3. The `level` member variable controls the depth of the look-ahead depth searching AI. The depth of the search determines the intelligence of the computer player because higher-depth searches result in a more intelligent computer player.
4. The `serviceRepaints()` method forces the canvas to be repainted, which is very useful when you need the screen to be repainted immediately.

Quiz Answers for Day 20

1. A J2ME server application is designed to run on a Web server, and typically facilitates the sale and distribution of end-user J2ME applications. End-user J2ME applications are designed to run on J2ME devices; MIDlets are good examples of end-user J2ME applications.
2. Application streaming is the process of delivering an application so that a user can begin using the application while it is still being downloaded. The significance of this is that it reduces the delay involved in downloading an application over a relatively slow wireless connection.
3. A J2ME portal is a Web site devoted to the distribution of end-user J2ME applications.

Quiz Answers for Day 21

1. The difference between the CLDC and the CDC is that the CDC is targeted at devices that have significantly more memory and processing power than CLDC devices.
2. The C virtual machine (CVM) is a full-featured Java virtual machine that supports all of the functionality of the Java language. The CDC uses the CVM, whereas the CLDC relies on the more limited KVM.
3. A smart card is a small plastic card the size of a credit card that houses a tiny computer capable of running compact applications.

APPENDIX B

What's on the CD-ROM

The CD-ROM accompanying this book includes a variety of resources that you will find useful as you go about developing MIDlets of your own. The contents of the CD-ROM can be broken down into the following major J2ME resources:

- Sample source code
- J2ME tools

Sample Source Code

Perhaps the most valuable resource on the CD-ROM is the collection of source code for all the sample MIDlets covered throughout the book. These are great examples of wireless J2ME development concepts, and this source code makes a great starting point for creating your own MIDlets. The source code for the following MIDlets is included on the CD-ROM:

- Howdy
- SysInfo

- Olympics
- SlideShow
- Mortgage
- Fortune
- Weather
- Directions
- ToDoList
- Contacts
- CheckRegister
- AuctionWatch
- Atoms
- TravelingGecko
- Connect4

J2ME Tools

The CD-ROM includes some important development tools that you'll likely find useful in MIDlet development: Sun's Forte 2.0 Community Edition and Sun's Java 2 SDK.

Forte for Java is a comprehensive visual development environment for J2SE and J2EE development. Forte for Java doesn't come standard with support for J2ME, but a special add-on module is included in the J2ME Wireless Toolkit that adds J2ME support to Forte. When the J2ME add-on module is installed, Forte for Java becomes a visual J2ME development environment that significantly improves the MIDlet development process.

Note

Sun's J2ME Wireless Toolkit can be downloaded from `http://java.sun.com/products/j2mewtoolkit/download.html`.

The Java 2 SDK included on the CD is a feature-complete development and deployment platform designed for the web. Java applications can be designed for and deployed on multiple operating systems and platforms.

Note

For additional sample MIDlet code, check out the Motorola J2ME Wireless Toolkit, available with the Motorola SDK for J2ME and which can be downloaded from the Motorola iDEN Developer Web site at `http://www.idendev.com`.

The following Web servers are included on the accompanying CD-ROM:

- Apache
- Enhydra

Apache is the most widely used open source Web server in existence as of this writing, and is developed by The Apache Software Foundation. Apache is a pure Web browser and doesn't directly address the needs of enterprise applications.

If you are developing a MIDlet as part of a larger enterprise application, you might want to consider Enhydra, which is a popular open-source application server maintained by Lutris Technologies. Enhydra plays a similar role to Apache, but its focus is centered more on e-commerce applications that require a full-blown application server.

INDEX

A

Active MIDlets, 65
addSprite() method (Sprite class), 417
addTransactionRecord() method, CheckRegister MIDlet, 348
AI (artifical intelligence), 462, 464-465
- Connect4 game, 467-468
- strategic AI, 464

Alert class, GUI and, 159-160
amount variable (check register), 345
anchor points, text and, 137
angles, arcs and, 134
animation, 403-404
- blue-screening, 406
- cast-based, 405-409
- cel animation, 408
- collision detection, 408-409
- frame-based, 405
- overview, 404-405
- sprite animation, 403
  - implementation, 409-427
- sprites, 406
- transparency, 407
- types, 405-409
- Z-order, 407-408

any constant, 163
APIs (Application Programming Interface)
- CLDC, 85, 87-92
- data type wrappers, 97-98
- MIDlets and, 86-87
- MIDP, 85
- MIDP API, 92-97

append() method, 156
applications, 502
- end-users, 504-506
- portals, 503-504
- server applications, 502-503

arcs, drawing, 134-135
artificial intelligence. *See* AI
Atoms example MIDlet, 427-433
AtomsCanvas class, 427-431
AuctionWatch MIDlet
- building, 377-396
- commands, 384-386
- retrieving live data, 386-390
- source code, 391-395

AuctionWatch() constructor, 383-384
AWT (Advanced Windowing Toolkit), 131
axes, animation sprites and, 407

B

Back command
- check register, 357
- Weather MIDlet, 223-224

BACK command constant, 176
Background class, 423-427
Background object, SpriteVector class, 419-420
Blackberry handhelds, 499
blocking, I/O and, 194
blue-screening, animation and, 406
Bluetooth, 16
bold text, 136
browsers, micro-browsers, 505
buffers, offscreen buffer, 131
bytecode verifier, 30

C

Calendar class, 98-99
CANCEL command constant, 176
canvasses, MIDlets, 66-67
cast-based animation, 405-409
CD-ROM bundle
- J2ME documentation, 533
- J2ME tools, 532-533
- source code, 531-532
- visual development environments, 532-533
- Web servers, 533

CDC (Connected Device Configuration), 510-511
cel animation, 408
character arrays, getChars( ) method, 163
characters, unprintable, 221
check register, electronic, 340
CheckRegister MIDlet
- building, 341-365
- commands, 353-358
- constructor, 351-353
- member variables, 349-351
- source code, 359-365
- testing, 365-369
- transaction records, 341-349

ChoiceGroup class, GUI and, 172-173
class file extension, MIDlets, 26
classes
- Alert, 159-160
- AtomsCanvas, 427-431
- Background, 423-427
- Calendar, 98-99
- ChoiceGroup, 172-173
- ColorBackground, 425-426
- Command, 175-177
- Connect4 game engine classes, 470-480
- Connect4Canvas, 481-486
- Connect4Engine, 470-473
- Connect4State, 473-480
- Contact, 314-316
- ContactDB, 316-320
- DateField, 170-171
- Displayable, 153
- Gauge, 173-175
- Graphics, 131-139
- Image, 138-139
- ImageBackground, 426-427
- ImageItem, 167-169
- InputStream, 194-195
- Item, 377-379
- ItemDB, 379-381
- List, 160-162
- MIDlet I/O, 193
- MIDP API, 93-97
- OutputStream, 195-196
- RecordStore (RMS), 284
- Screen, 155
- Sprite, 410-418
- SpriteVector, 418-423
- SSCanvas, 143-148
- StringItem, 165-167
- TextBox, 163-164
- TextField, 169-170
- Ticker, 158, 164-165
- Transaction, 341-346
- TransactionDB, 341, 346-349

CLDC (Connected Limited Device Configuration), 16-18, 510-511
- Java and, 18-19
- security and, 19

CLDC API, 85-92
- GCF (Generic Connection Framework), 91-92
- interfaces, 88
- J2SE API and, 88-90

CLDC ASI
- java.io package, 90
- java.lang package, 89
- java.util package, 89-90

closing record stores, 289
code, unprintable characters, 221
CodeWarrier for Java, 36

coding, MIDlets, 69-73
collision detection
animation and, 408-409
sprites, 417-418
Traveling Gecko game, 449-450
collision( ) method, SpriteVector class, 421
color, 129-131
AWT, 131
Configuration Editor, 113-115
HSB system, 130
intensities, 130
numeric values, 130
RGB system, 130
transparent, 407
ColorBackground class, 425-426
Command class, 175-177
command line, running emulator, 45-52, 54-55
commandAction( ) method, 225, 386
commands
AuctionWatch MIDlet, 384-386
CheckRegister MIDlet, 353-358
Contacts MIDlet, 323-326
MIDlets, 66
MIDP GUI components, 175-177
priorities, 176
Weather MIDlet, 224-225
compiling MIDlets, 74-75
computerMove( ) method, 471
computers, hand-held, 499-501
configuration
CLDC, 16-18
devices. *See* device profiles
Configuration Editor (Motorola SDK), 32-33, 107-118
color, 113-115
device profiles, 107-118
Events tab, 117
KVM tag, 115-116
networking, 116-117
Wireless Toolkit, 30
Connect4 game, 461
AI strategy, 467-468
constructing, 469, 471-487
design, 465-468
engine classes, 470-480
MIDlet class, 486-487
testing, 488
user interface, 466-467
Connect4Canvas class, 481-486
Connect4Engine class, 470-473
Connect4State class, 473-480
Connection interface, 190
connection strings, networks, 192
connections, networks, 196-198
Connector class
interfaces, 190
networking and, 191
constants
any, 163
emailaddr, 163
numeric, 163
password, 163
phonenumber, 163
URL, 163
constructors
AuctionWatch MIDlet, 383-384
CheckRegister MIDlet, 351-353
Sprites, 412-413
Weather( ), 223-224
Contact class, 314-316
ContactDB class, 316-320
Contacts, 311
overview, 312-313
Contacts MIDlet, 313-334
ContentConnection interface, 190
CVM (C virtual machine), 511

D

data type wrappers, APIs, 97-98
DatagramConnection interface, 190
date and time, 98-99
DateField class, GUI and, 170-171
Delete command, check register, 357
delete( ) method, 156
deleteTransactionRecord( ) method, CheckRegister MIDlet, 348
deleting records from record stores, 287
design
Connect4 game, 465-468
Traveling Gecko game, 438-440

Destroyed MIDlets, 65
development
MIDlets, 67-68
overview, 26-29
development environments, 33-36
development kit, 25-26
device profiles
Configuration Editor, 107-118
custom, 105-106
testing, 120-121
display properties, 108-115
events, 117
execution speed, 115-116
key mappings, 108-115
memory, 115-116
networking, 116-117
overview, 106-107
tweaking, 118-119
devices
J2ME, 494-501
JPEG images, 109-110
Directions MIDlet, 248-259
testing, 259-261
display
MIDlets, 66-67
MIDP devices, 21
properties, Configuration Editor, 108-115
Display object, MIDlets, 152
Displayable class, 153
forms, 154-157
screens, 154-157
displayable objects, MIDlets, 152
displays, MIDlets, 152-154
documentation, J2ME CD-ROM bundle, 533
draw( ) method
Sprite class, 416-417
SpriteVector class, 422-423
drawArc( ) method, 134
drawLine( ) method, 133
drawRect( ) method, 133
drawString( ) method, 137
driving directions. *See also* Directions MIDlet
extracting, 243-247

E

eBay, WAP and, 375
Edit Key Mappings (Configuration Editor), 110-111
electronic check register, 340
transaction types, 341
emailaddr constant, 163
emulator, 31-32
command line, 45-52, 54-55
Forte and, 58, 60
KToolbar and, 56-58
MIDlets, examples, 48-49
no script, 49-51
physical devices, 44-45
run.bat script, 46-48
screens, 45
visual environments, 56-58, 60
emulators, 28
device profiles and, 106
Motorola SDK for J2ME, 51-52, 54-56
end-user applications, 504-506
enumerateTransaction-Record( ) method, CheckRegister MIDlet, 348
enumerating records, RMS, 288-289
evaluations, optimization and, 272
Events tag, Configuration Editor, 117
example MIDlets
Fortune, 198-209
mortgage, 178-186
ToDoList, 289-303, 305-307
exclusive lists, 160
execution speed, device profiles, 115-116
Exit command, Weather MIDlet, 223-224
EXIT command constant, 176
expressions/subexpressions, optimization and, 273

F

fillArc( ) method, 135
financial applications, 339-341
flush( ) method, output stream, 196
fonts, 136
forms
defined, 156
Displayable class, 154-157
GUI components, 156
indexes, 156
Forte, 58, 60
Forte for Java, 34-35

Fortune MIDlet example, 198-209
foundation profile, 512
fps (frames per second), animation and, 404
frame-based animation, 405-406
frames, images (Sprite class), 410
frames per second (fps), animation and, 404
future of J2ME, 509-518

G

games, 437-438, 461. *See also* Traveling Gecko
AI and, 462, 464-465
Connect4, 461, 465-468
Sega and, 516
gaming, Java and, 107
gas pump applications, 514-515
Gauge class, GUI and, 173-175
GCF (Generic Connection Framework), 91-92, 190
getChars( ) method, 163
getConditions( ) method, 225
getCurrent( ) method, Display class and, 154
getFont( ) method, 137
getLiveItemInfo( ) method, AuctionWatch MIDlet, 387
getSelectedFlags( ) method, 161
getTicker( ) method, 155
getTransactionRecord( ) method, CheckRegister MIDlet, 348
Go command, Weather MIDlet, 223-224
graphical MIDlets, building, 139-142
graphical objects
animation and, 405
frame-based animation and, 405
graphics
arcs, 134-135
color, 129-131
images, 138-139
lines, 132-133
MIDP graphics coodinate system, 128-129
offscreen buffer, 131
primitives, 132-135
rectangles, 133-134
text and, 135-138
Graphics class, 131-139
Graphics.DOTTED, 133
Graphics.SOLID, 133
GUIs (graphical user interfaces). *See also* MIDlet GUIs
AuctionWatch MIDlet, 382
MIDP components, 158-175
commands, 175-177

H-I

handheld computers, Java, 499-501
hardware requirements
CLDC, 18
MIDP, 20-21
HELP command constant, 176
homes, smart homes, 514
Howdy MIDlet, 68-81
HSB color, 130
HttpConnection interface, 191
I/O (input/output) MIDlet. *See* MIDlet I/O
Image class, 138-139
ImageBackground class, 426-427
ImageItem class, GUI and, 167-169
images, 138
drawing, 138-139
frame images, Sprite class, 410
implicit lists, 160
incFrame( ) method, Sprite class, 413-414
indexes
forms, 156
lists, 162
input, MDIP devices, 21
InputConnection interface, 190
InputStream class, 194-195
insert( ) method, 156
interactivity, text, 163
interfaces
CLDC API, 88
Connector class, 190
HttpConnection, 191
MIDP API, 93-97
RMS, 284-285
Weather MIDlet, 222-224
intersects( ) method, Sprites, 418
italic text, 136

Item class
  AuctionWatch MIDlet, 377-379
  GUI components and, 156
ITEM command constant, 176
ItemDB class, AuctionWatch MIDlet, 377, 379-381
ITU-T key set, 45

J

J2ME
  devices, 494-501
  documentation, CD-ROM bundle, 533
  emulator, 41-45
    device profiles and, 106
  future of, 509-518
  overview, 9-22
  tools, CD-ROM bundle, 532-533
J2SE (Java 2 Special Edition), 13
J2SE API
  CLDC API and, 88-90
  MIDP API and, 92-93
JAR files, MIDlets, 27
Java
  CLDC and, 18-19
  gaming and, 107
Java 2 SDK, 29
Java Card, 513-514
Java-powered mobile phones, 494-497
java.io package, CLDC ASI and, 90
java.lang package, CLDC ASI and, 89
java.lang.Runtime class, 98
java.util package, CLDC ASI and, 89-90
javax.microedition.io package, MIDP API, 95
javax.microedition.lcdui package, 158
  MIDP API, 93-95
javax.microedition.midlet package, MIDP API, 93
javax.microedition.rms package, MIDP API, 96-97
Jbuilder Handheld Express, 36
Jini, 512-513
JPEG images, devices, 109-110

K-L

Kbrowser (4thpass), 505-506
key mappings, Configuration Editor, 108-115
KToolBar, 33-34
KToolbar develoment environment, 56-58
  Howdy MIDlet and, 79-81
KVM (K Virtual Machine), 26, 511
KVM tab, Configuration Editor, 115-116
lifecycle methods, MIDlets, 65
lines, drawing, 132-133
List class, GUI and, 160-162
List() constructor, 161
lists
  exclusive, 160
  implicit, 160
  indexes, 162
  multiple, 160-161
literals, converting to strings, 97
local variables, optimization and, 273
loop expansion, optimization and, 273-274

M

maintainability, optimization and, 267
makeMove() method, 471
MapBlast.com, 238, 241
MapQuest.com, 241-242
member variables, CheckRegister MIDlet, 349-351
memory
  device profiles, 115-116
  Java Card, 513-514
  optimization and, 269-271
  smart cards, 513-514
methods
  commandAction(), 225, 386
  drawArc(), 134
  drawLine(), 133
  drawRect(), 133
  getConditions(), 225
  getFont(), 137
  InputStream class, 194
  ItemDB class, 381
  makeMove(), 471
  paint(), 131
  Sprite class, 413

micro-browsers, 505
MIDlet
MIDlet class, Connect4 game, 486-487
MIDlet GUIs, 151
MIDlet I/O, 193-196
MIDlet Suites, 27
MIDlets, 26
- APIs and, 86-87
- canvases, 66-67
- coding, 69-73
- commands, 66
- compiling, 74-75
- Contacts, 313-334
- development, 67-68
- Directions, 248-259
  - testing, 259-261
- display, 66-67, 152-154
- Display object, 152
- displayable objects, 152
- drive directions, extracting, 243-247
- examples, 48-49
  - Atoms, 427-433
  - Fortune, 198-209
  - Motorola SDK for J2ME, 55-56
- graphical, building, 139-142
- Howdy, 68-79
- I/O classes, 193
- overview, 64-65
- JAR files, 27
- lifecycle, 66
- lifecycle methods, 65
- packaging, 75-77
- pre-verification, 75
- samples
  - mortgage sample, 178-186
  - ToDoList, 289-303, 305-307
- screens, 66-67
- slideshow, 143-148
- states, 65
- SysInfo, 99-103
- testing, 78-79
- weather, 221
  - live, 215
  - user interface, 222-224

MIDP (Mobile Information Device Profile), 19-20, 92
- graphics coordinate system, 128-129
- networking, 190-193

MIDP API, 85, 92-97
- classes, 93-97
- HttpConnection interface, 191
- interfaces, 93-97
- J2SE API and, 92-93
- javax.microedition.io package, 95
- javax.microedition.lcdui package, 93-95
- javax.microedition.midlet package, 93
- javax.microedition.rms package, 96-97

MIDP graphics API, color, 129-131
MIDP GUI components, 158-175
- commands, 175-177

mobile phones, Java-powered, 494-497
mobile spending, 340-341
MobileDASH (AGEA), 504-505
mortgage sample MIDlet, 178-186
Motorola SDK, Configuration Editor, 107-118
Motorola SDK for J2ME, 51-52, 54-56
movies, frame-based animation and, 405
MP3s, Sensate and, 517
multimedia, 517-518
multiple lists, 160-161
musical phones, 517

N-O

navigation, Web and, 238, 241-242
networking
- connection strings, 192
- connections, 196-198
- Connector class, 191
- device profiles, 116-117
- MIDP, 21
  - overview, 190-193
- Jini, 512-513

numeric constant, 163
numeric values, converting, 98

offscreen buffer, 131
OK command constant, 176
online auctions, 373-376. *See also* **AuctionWatch MIDlet**
openRecordStore( ) method, 285
operating systems, 21-22
optimization, 265
- basic principles, 266-268
- evaluation and, 272
- local variables and, 273
- loop expansion and, 273-274
- maintainability and, 267

memory usage and, 269-271
minimization and, 269
significance of, 274-275
simplification and, 269
size, 267-268
speed, 268
subexpressins and, 273
techniques for, 271-274
optimized MIDlets. *See* optimization
OutputConnection interface, 190
OutputStream class, 195-196
ovals, arcs and, 134

P

packages
javax.microedition.lcdui, 158
MIDlets, 75-77
RMS, 283-285
pagers, 497-499
paint( ) method, 131
Palm handheld computers, 499-501
password constant, 163
Paused MIDlets, 65
PDAs (personal digital assistants), 499-501
persistent storage, 282
PersonalJava, 512
phonenumber constant, 163
phones, musical, 517
pixels, arcs and, 135
portals, application portals, 503-504
primitives, graphics, 132-135

Q-R

record ID, RMS, 283
record stores, RMS, 283
adding records, 286-287
closing, 289
deleting records, 287
enumerating records, 288-289
opening, 285-286
retrieving records, 287
records
Contacts, 314-320
RMS, 283
RecordStore class (RMS), 284
rectangles
drawing, 133-134
rounded rectangles, 133
RGB color, 130
RIM (Research In Motion), 497
RMS (Record Management System)
interfaces, 284-285
overview, 281-283
persistent storage, 282
record ID, 283
record stores, 283
adding records, 286-287
closing, 289
deleting records, 287
enumerating records, 288-289
opening, 285-286
retrieving records, 287
records, 283
RecordStore class, 284
RMS package, 283-285
rounded rectangles, 133
run.bat script, emulator, 46-48
runtime environment, 98

S

Save command, check register, 358
Scheme parameter, network protocol, 192
Screen class, 155
SCREEN command constant, 176
screens
AuctionWatch MIDlet, 382
Displayable class, 154-157
emulator and, 45
MIDlets, 66-67
SDK, 29
security, CLDC, 19
Sega, J2ME games and, 516
Select command, check register, 357
Sensate, MP3s and, 517
server applications, 502-503
setColor( ) method, Graphics class, 131
setCurrent( ) method, Display class and, 154
setCurret( ) method, Displayable objects and, 154
setTicker( ) method, 155
setTransactionRecord( ) method, CheckRegister MIDlet, 348
size optimization, 267-268
slideshow MIDlets, 143-148
smart cards, 513-514

smart homes, 514
SMS (Short Messaging Service), 15
source code
  AuctionWatch MIDlet, 391-395
  CD-ROM bundle, 531-532
  CheckRegister MIDlet, 359-365
space exploration, 515-516
speed optimization, 268
spending, tracking. *See* mobile spending
sprite animation, 403. *See also* cast-based animation
  implementation, 409-427
Sprite class, 410-418
  Traveling Gecko, 442-450
Sprite constructors, 412-413
sprites
  animation, 406
  axes and, 407
  collision detection, 408-409
  Traveling Gecko game, 438-440
  XY position, 410
SpriteVector class, 418-423
SSCanvas class, 143-148
state changes, responding to, 157
STOP command constant, 176
strategic AI, 464
  weighting, 464-465
StreamConnection interface, 191
StreamConnectionNotifier interface, 191
StringItem class, GUI and, 165-167
strings
  connection strings, networks, 192
  converting from literals, 97
subexpressions, optimization and, 273
SysInfo MIDlet, 99-103

T

TalkAbout radios, 495
Target parameter, network protocol, 192
television, frame-based animation and, 406
testCollision( ) method, Sprite class, 417-418
testing
  AuctionWatch MIDlet, 396, 398
  CheckRegister MIDlet, 365-369
  device profiles, 120-121
  Directions MIDlet, 259-261
  MIDlets, 78-79
  Traveling Gecko game, 456-457
  Weather MIDlet, 232
text
  anchor points, 137
  drawing, 135-138
  fonts, 136
  interactivity, 163
TextBox class, GUI and, 163-164
TextField class, GUI and, 169-170
TGCanvas class, Traveling Gecko game, 450-454
Ticker class, 158, GUI and, 164-165
ToDoList MIDlet sample, 289-303, 305-307
Transaction class (check register), 341-346
transaction records, CheckRegister MIDlet, 341-349
transaction types, check register, 341
TransactionDB class (check register), 341, 346-349
transparency, animation and, 407
transparent color, 407
Traveling Gecko game
  construction, 440-456
  design, 438-440
  Gecko class, 442-445
  Geckocide class, 445-446
  Rock class, 447-448
  Sprite classes, 442-450
  sprites, 438-440
  testing, 456-457
  TGCanvas class, 450-454
  TGVector class, 449-450

U-V

unprintable characters, 221
update( ) method, Sprite class, 414-416
URL constant, 163
user interface
  AuctionWatch MIDlet, 381-384
  Connect4 game, 466-467
  Contacts MIDlet, 320-323

variables
- local, optimization and, 273
- Sprite class, 410-411

verification, pre-verification, 26, 75
video, 517
virtual machine, bytecode verifier, 30
visual development environments, 33-36
- CD-ROM, 532-533

W-Z

WAP (Wireless Application Protocol), 15
- eBay and, 375

weather, 214-215, 217-221
Weather MIDlet, 221
- commands, 224-225
- source code, complete, 228-232
- testing, 232
- user interface, 222-224
- weather conditions, retrieving, 226-228

Weather( ) constructor, 223-224
Weather.com Web site, 214
Weather.gov Web site, 215, 217
Web navigation, 238, 241-242
Web servers, 36-37
- CD-ROM bundle, 533

Web sites
- MapBlast.com, 238, 241
- MapQuest.com, 241-242
- Weather.com, 214
- Weather.gov, 215, 217

weighting, strategic AI, 464-465
wireless computing, 10-13
Wireless Toolkit, 29-33
WML (Wireless Markup Language), 15

XY position, sprites, 410

Z-order, animation and, 407-408

JAVA™ 2 SOFTWARE DEVELOPMENT KIT STANDARD EDITION VERSION 1.3 SUPPLEMENTAL LICENSE TERMS

These supplemental license terms ("Supplemental Terms") add to or modify the terms of the Binary Code License Agreement (collectively, the "Agreement"). Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Agreement. These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Agreement, or in any license contained within the Software.

1. **Internal Use and Development License Grant**. Subject to the terms and conditions of this Agreement, including, but not limited to, Section 2 (Redistributables) and Section 4 (Java Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce the Software for internal use only for the sole purpose of development of your Java™applet and application ("Program"), provided that you do not redistribute the Software in whole or in part, either separately or included with any Program.

2. **Redistributables**. In addition to the license granted in Section 1 above, Sun grants you a nonexclusive, non-transferable, limited license to reproduce and distribute, only as part of your separate copy of JAVA™ 2 RUNTIME ENVIRONMENT STANDARD EDITION VERSION 1.3 software, those files specifically identified as redistributable in the JAVA™ 2 RUNTIME ENVIRONMENT STANDARD EDITION VERSION 1.3 "README" file (the "Redistributables") provided that: (a) you distribute the Redistributables complete and unmodified (unless otherwise specified in the applicable README file), and only bundled as part of the Java™ applets and applications that you develop (the "Programs:); (b) you do not distribute additional software intended to supersede any component(s) of the Redistributables; (c) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables; (d) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, and (e) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts, and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

3. **Separate Distribution License Required**. You understand and agree that you must first obtain a separate license from Sun prior to reproducing or modifying any portion of the Software other than as provided with respect to Redistributables in Section 2.

4. **Java Technology Restrictions**. You may not modify the Java Platform Interface ("JPI", identified as classes contained within the "java" package or any subpackages of the "java" package), by creating additional classes within the JPI or otherwise causing the addition to or modification of the classes in the JPI. In the event that you create an additional class and associated API(s) which (i) extends the functionality of a Java environment, and (ii) is exposed to third party software developers for the purpose of developing additional software which invokes such additional API, you must promptly publish broadly an accurate specification for such API for free use by all developers. You may not create, or authorize your licensees to create additional classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun", or similar convention as specified by Sun in any class file naming convention. Refer to the appropriate version of the Java Runtime Environment binary code license (currently located at `http://www.java.sun.com/jdk/index.html`) for the availability of runtime code which may be distributed with Java applets and applications.

5. **Trademarks and Logos**. You acknowledge and agree as between you and Sun that Sun owns the Java trademark and all Java-related trademarks, service marks, logos, and other brand designations including the Coffee Cup logo and Duke logo ("Java Marks"), and you agree to comply with the Sun Trademark and Logo Usage Requirements currently located at `http://www.sun.com/policies/trademarks`. Any use you make of the Java Marks inures to Sun's benefit.

6. **Source Code**. Software may contain source code that is provided solely for reference purposes pursuant to the terms of this Agreement.

7. **Termination**. Sun may terminate this Agreement immediately should any Software become, or in Sun's opinion be likely to become, the subject of a claim of infringement of a patent, trade secret, copyright or other intellectual property right.

JAVA™ DEVELOPMENT TOOLS FORTE™ FOR JAVA™, RELEASE 2.0, COMMUNITY EDITION SUPPLEMENTAL LICENSE TERMS

These supplemental license terms ("Supplemental Terms") add to or modify the terms of the Binary Code License Agreement (collectively, the "Agreement"). Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Agreement. These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Agreement, or in any license contained within the Software.

1. **Software Internal Use and Development License Grant**. Subject to the terms and conditions of this Agreement, including, but not limited to Section 3 (Java™ Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce internally and use internally the binary form of the Software complete and unmodified for the sole purpose of designing, developing and testing your [Java applets and] applications intended to run on the Java platform ("Programs").
2. **License to Distribute Redistributables**. In addition to the license granted in Section 1 (Redistributables Internal Use and Development License Grant) of these Supplemental Terms, subject to the terms and conditions of this Agreement, including but not limited to Section 3 (Java Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce and distribute those files specifically identified as redistributable in the Software "README" file ("Redistributables") provided that: (i) you distribute the Redistributables complete and unmodified (unless otherwise specified in the applicable README file), and only bundled as part of your Programs, (ii) you do not distribute additional software intended to supersede any component(s) of the Redistributables, (iii) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables, (iv) for a particular version of the Java platform, any executable output generated by a compiler that is contained in the Software must (a) only be compiled from source code that conforms to the corresponding version of the OEM Java Language Specification; (b) be in the class file format defined by the corresponding version of the OEM Java Virtual Machine Specification; and (c) execute properly on a reference runtime, as specified by Sun, associated with such version of the Java platform, (v) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, and

What’s on the CD-ROM

The companion CD-ROM contains Sun Microsystem’s Java Software Development Kit (SDK) version 1.3 and Forte 2.0 Community Edition, plus the example code from the book.

Windows 95, 98, ME, Windows NT 4.0, or Windows 2000 Installation Instructions

1. Insert the disc into your CD-ROM drive.
2. From the Windows desktop, double-click the My Computer icon.
3. Double-click the icon representing your CD-ROM drive.
4. Double-click the icon titled START.EXE to run the installation program.
5. Follow the onscreen prompts to finish the installation.

Note

If you have the AutoPlay feature enabled, the START.EXE program starts automatically whenever you insert the disc into your CD-ROM drive.

Linux and Unix Installation Instructions

These installation instructions assume that you have a passing familiarity with Unix commands and the basic setup of your machine. As Unix has many flavors, only generic commands are used. If you have any problems with the commands, please consult the appropriate manual page or your system administrator.

Insert CD-ROM in CD drive.

If you have a volume manager, mounting of the CD-ROM will be automatic. If you don’t have a volume manager, you can mount the CD-ROM by typing `mount-tiso9660/dev/cdrom/mnt/cdrom`.

Note

`/mnt/cdrom` is just a mount point, but it must exist when you issue the mount command. You may also use any empty directory for a mount point if you don’t want to use `/mnt/cdrom`.

Open the readme.txt file for descriptions and installation instructions.